Parcours paralleles de graphes d'etats par des algorithmes de la famille de A* en intelligence artificielle

Van-Dat Cung

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Parcours parallèles de graphes d'états par des algorithmes de la famille de A en intelligence artificielle*

Van-Dat CUNG
Catherine ROUCAIROL

N° 1900
Mai 1993

PROGRAMME 1

Architectures parallèles,
Bases de données,
Réseaux et Systèmes distribués

Parcours Parallèles de Graphes d'États par des algorithmes de la famille de A* en Intelligence Artificielle

---

Parallel and Distributed A*-like State Space Search Algorithms in Artificial Intelligence

Van-Dat CUNG et Catherine ROUCAIROL

Avril 1993

Laboratoire MASI
Université Versailles - Saint Quentin en Yvelines
45, avenue de États-Unis
78000 VERSAILLES, FRANCE

INRIA - Action Paradis
Domaine de Voluceau - BP 105 Rocquencourt
78153 LE CHESNAY, FRANCE

e-mail:
Van-Dat.Cung@inria.fr
cung@masi.ibp.fr

Résumé

De nombreux algorithmes parallèles pour les parcours de graphes d'états de la famille de A* en Intelligence Artificielle, implantés sur des machines allant du type SIMD massivement parallèle au réseau de stations de travail en passant par le MIMD à mémoire partagée, ont été proposés cette dernière décennie.

Ces algorithmes sont analysés et classifiés suivant le type de parallèlisme et l'architecture des machines parallèles utilisées.

Nous avons dégagé et répertorié les principales techniques de parallélisation des parcours de graphes d'états en Intelligence Artificielle et en Recherche Opérationelle, ainsi que leurs modes de programmation.

Abstract

In Artificial Intelligence, various parallelization of A*-like state space search algorithms have been proposed during the last decade. They are implemented on different parallel machines from massively parallel SIMD to multi-processors with shared memory and distributed networks.

These algorithms are analyzed and classified following the kind of the parallelism and the architecture of the parallel machines concerned.

We point out the main parallelization techniques used for state space search in Artificial Intelligence and in Operational Research together with their programming schemes.

Centres d'intérêt.

Parcours de graphes d'états, A* et variantes, algorithmique parallèle et distribuée, machines SIMD massivement parallèle et MIMD.

Keywords.

State space search algorithms, A*-like, parallel and distributed algorithms, massively parallel SIMD and MIMD parallel machines.

Table des matières

Table des figures

Introduction

Un des objectifs importants, commun aux domaines de l'Intelligence Artificielle (IA) et de la Recherche Opérationnelle (RO), est la recherche (d'une ou) des solutions, exactes ou approchées, de problèmes de décision ou de problèmes d'optimisation difficile (de la classe NP-complet).

Citons pour exemples :

- le voyageur de commerce, le sac à dos;
- le placement des circuits intégrés sur une carte, en CAO des VLSI;
- la programmation logique;
- les jeux comme les échecs et le taquin.

Cette recherche de solutions nécessite généralement le parcours d'un espace, communément appelé *espace de recherche*[1]. Cet espace de recherche est représenté soit par un *graphe implicite d'états* en IA, soit par une *arborescence de sous-problèmes disjoints* en RO. Nous nous intéressons ici essentiellement au premier cas. Notons que des efforts ont été portés par Kumar et al., pour unifier les algorithmes de recherche de ces deux domaines [Nau 1984].

L'espace de recherche en IA peut en effet être considéré comme un graphe implicite d'états. Une instance donnée d'un problème, par exemple une configuration de départ du jeu de taquin (cf. Figure 0.1), correspond à un *état initial*, tandis qu'une solution du problème, par exemple les tuiles numérotées qui sont alignées au jeu de taquin, est un *état final* ou *état solution*. Le parcours de l'espace de recherche consiste donc à engendrer et à évaluer les états intermédiaires menant d'un état initial à un état final. Ces états constituent alors l'ensemble des sommets d'un graphe, et les changements qui permettent de passer un état à un autre correspondent aux arcs du graphe.

Or la cardinalité de l'ensemble d'états est généralement d'une taille trop importante pour que l'on puisse les connaître a priori. Cette contrainte est traduite au niveau des algorithmes par leur *complexité spatiale* et *temporelle*. Les états sont engendrés au fur et à mesure du parcours du graphe qui est donc implicite.

Aussi pour éviter une exploration exhaustive, les algorithmes de parcours implicites de graphes doivent donc tenir compte des informations accumulées pendant la recherche afin d'élaguer les états jugés sans intérêt pour la résolution du problème. Ces états ne sont jamais engendrés et évalués pendant le parcours du graphe.

[1] Search space en anglais.

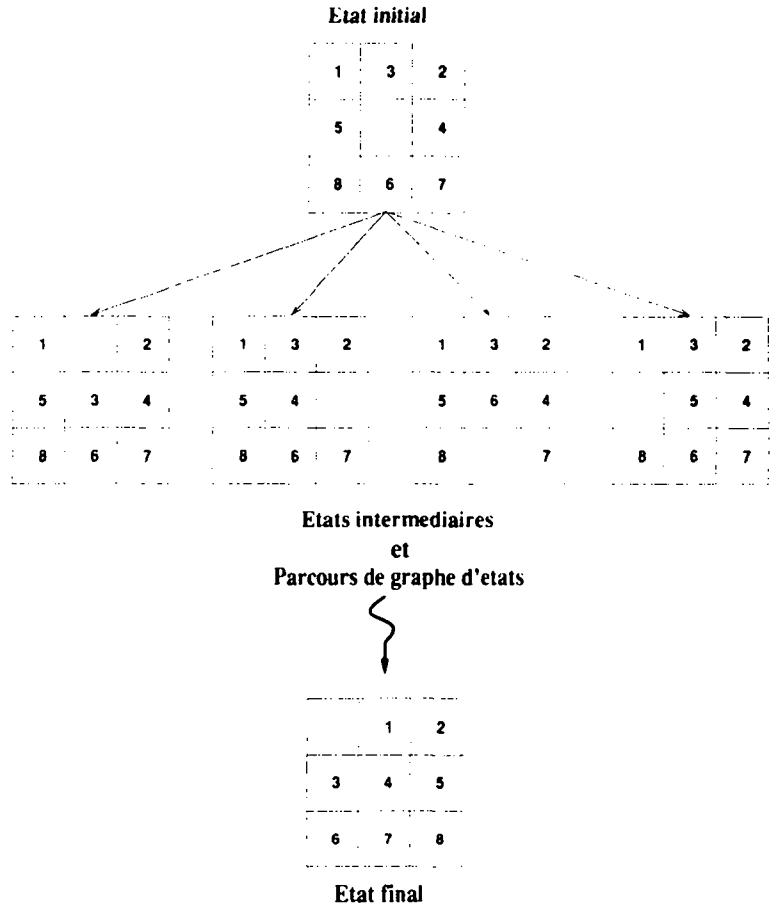

Figure 0.1 : Graphe d'états du jeu de taquin.

Par la suite, nous nommerons indifféremment l'espace de recherche, par souci de simplification, graphe d'états ou graphe implicite d'états.

Dans ce rapport, nous étudions particulièrement les parallélisations des algorithmes de parcours de graphes d'états de la famille de A* (A* [Nilsson 1980], IDA* [Korf 1985], et RA* [Evett 1990]) utilisés en Intelligence Artificielle. Ces algorithmes ont tous la caractéristique de trouver un état final s'il existe, à partir d'un état initial et à travers un parcours de graphe d'états. Cet état est alors une solution dite *optimale* en ce sens que la longueur du chemin joignant l'état initial à l'état final est la plus courte possible.

Pour ce faire, ces algorithmes utilisent pour les évaluations des états une fonction d'évaluation f de la forme $g + h$, où g est la fonction donnant le coût du chemin de l'état initial à un état courant. La fonction h, appelée *fonction heuristique* ou *heuristique* tout simplement, estime le coût du chemin entre l'état courant et un état final. Un état final optimal est nécessairement trouvé s'il existe et si la fonction f est monotone et minorante [Nilsson 1980].

De nombreux auteurs [Korf 1985, Chakrabarti 1989, Ibaraki 1990, Evett 1990], ont travaillé sur la complexité spatiale de ces algorithmes. Des dérivées de A* ont été ainsi introduites pour limiter la croissance du graphe à explorer (IDA*, RA*), mais la compexité

temporelle de ces algorithmes séquentiels reste exponentielle et demeure un problème ouvert. L'idée d'utiliser les machines parallèles pour endiguer la croissance exponentielle de ce facteur clé est donc alors naturelle. Elle permettrait d'obtenir des temps de résolution raisonnables pour les applications, et peut être de traiter des problèmes de taille supérieure.

Le fait que les machines parallèles soient maintenant à des prix abordables pour les entreprises, donne une raison supplémentaire de développer dorénavant des algorithmes parallèles efficaces, exploitant toute la puissance offerte par ces machines.

La parallélisation des algorithmes de la famille de A*, vise deux objectifs :

1. améliorer les performances des applications d'Intelligence Artificielle en proposant des algorithmes parallèles nouveaux et meilleurs en temps d'exécution.

2. proposer de nouveaux algorithmes séquentiels plus performants.

Le deuxième objectif mérite un peu d'explications. En 1978, Baudet [Baudet 1978] a découvert pendant ses études de parallélisation de l'algorithme $\alpha - \beta$ [Knuth 1975] qu'il était possible de l'améliorer en réduisant la fenêtre de recherche $[\alpha, \beta]$ initiale. Cette remarque a suscité par la suite de nombreuses études et propositions d'algorithmes séquentiels améliorant $\alpha - \beta$ [Campbell 1983, Pearl 1984, Marsland 1985, Reinefeld 1985, Weill 1992], toutes fondées sur les propriétés de cette fenêtre. C'est ainsi que nous étudierons l'exploitation en séquentiel des idées nouvelles que la conception d'un algorithme parallèle nous amènera.

Le premier chapitre de ce rapport est consacré aux diverses méthodes de parallélisation possibles des parcours de graphes d'états.

En raison de la multitude et de la diversité des algorithmes parallèles proposés dans la littérature, nous les avons classés dans les chapitres 2 et 3 en deux grandes familles suivant l'architecture de machines parallèles exploitée : MIMD et SIMD massif.

Cette étude nous permettra d'identifier les grandes lignes à suivre lors de la conception d'algorithmes de parcours de graphes d'états pour des machines à parallèlisme massif (CM-5, KSR, Meganode, Paragon) déjà accessibles dans les centres de recherche (IMAG, INRIA, IPG)

Chapitre 1

Parallélisme et parcours de graphes d'états

Le premier problème posé par la parallélisation des algorithmes de parcours de graphes d'états est la recherche du parallélisme sous-jacent dans les algorithmes séquentiels.

Rappelons d'abord la structure générale d'un algorithme séquentiel de parcours de graphes d'états :

```
PROCEDURE Parcours_de_Graphes (Ensemble_etats,
                               Intervalle_recherche);
1. Inserer_Etat (Ensemble_etats, Etat_initial);
   { Mettre l'etat initial courant dans un ensemble
     d'etats a explorer qui est initialement vide. }
2. Etat_courant := Choisir_Etat (Ensemble_etats);
   { Extraire un etat, eventuellement avec un ordre de priorite,
     de l'ensemble des etats; cet etat devient l'etat courant
     a explorer. }
3. TANTQUE (Etat_courant <> Etat_vide) ET
          (Etat_courant <> Etat_final) FAIRE
4.       Engendrer_Fils (Etat_courant, Ensemble_fils);
         { Les etats fils de l'etat courant sont engendres et
           mis dans un ensemble des etats fils. }
5.       Evaluer_Fils (Ensemble_fils);
         { Chaque element de cet ensemble d'etats fils est
           evalue a l'aide d'une fonction d'evaluation f et
           etiquete avec la valeur resultante de l'evaluation. }
6.       Mise_A_Jour (Intervalle_recherche);
         { Mettre a jour les bornes d'un intervalle de recherche
           avec les nouvelles valeurs obtenues a l'etape 5. }
7.       Elaguer_Etats (Ensemble_etats,
                        Ensemble_fils,
                        Intervalle_recherche);
         { Eliminer de l'ensemble des etats a explorer et
           de l'ensemble des etats fils les etats dont les
```

```
          valeurs d'evaluation ne sont pas
          dans l'intervalle de recherche. }
8.      Fusion_Ensembles (Ensemble_etats, Ensemble_fils);
        { On ajoute a l'ensemble des etats fils a l'ensemble
          des etats a explorer. }
9.      Etat_courant := Choisir_Etat (Ensemble_etats);
10. FIN_TANTQUE;
11. RETOURNER (Resultats);
    { Les resultats correspondent a l'etat final et
      au chemin le plus court.}
```

L'analyse de cet algorithme générique nous permet de dégager trois méthodes de parallélisation possibles :

1. la parallélisation du traitement d'un état,
2. la décomposition du parcours de graphes d'états,
3. le découpage de l'intervalle de recherche.

Nous pouvons remarquer que ces parallélisations portent justement sur les trois opérations de base des algorithmes de parcours de graphes d'états, à savoir les opérations qui consistent respectivement à :

1. engendrer les états fils d'un état courant et à les évaluer avec une fonction (étape 4. et 5.),
2. parcourir successivement les états d'un graphe (la boucle TANTQUE),
3. comparer les valeurs des états évalués par rapport (à une ou) aux bornes de l'intervalle de recherche afin d'effectuer les élagages des états inutiles (étape 6. et 7.).

1.1 Parallélisation du traitement d'un état

Le traitement d'un état consiste en une génération des états fils si l'état considéré n'est pas final, et en une évaluation de ces états à l'aide d'une fonction heuristique.

La parallélisation de ce traitement s'appuie d'une part sur la séparation des opérations de génération et d'évaluation des états, et d'autre part sur la décomposition de ces deux opérations en opérations élémentaires afin d'utiliser les techniques de «pipeline», de «pattern-matching» et de VLSI. Cette approche est employée notamment dans les programmes de jeux d'échecs comme HITECH [Berliner 1989] et DEEP THOUGHT [Hsu 1990].

Cependant cette idée présente l'inconvénient d'être très spécifique au domaine de l'application considérée, et manque par conséquent de généralité et de souplesse dans la mise en œuvre. Aussi, les performances espérées en terme d'accélération, par ce type de parallélisation sont faibles, car l'exploration des graphes d'états reste séquentielle. De meilleures accélérations, même faibles, devraient être obtenues pour les applications où la génération et l'évaluation des états sont difficiles et coûteuses.

1.2 Décomposition du parcours du graphe d'états

Le parcours du graphe d'états est décomposé en parcours de sous-graphes d'états. Différents processus peuvent être ainsi affectés à leur exploration suivant des stratégies à préciser.

La stratégie d'affectation d'états aux processus peut consister à affecter, soit un sous-graphe d'états à un processus [Marsland 1982, Kumar 1987, Ferguson 1988, Powley 1989, Li 1991], *parallélisation verticale* [Roucairol 1990], soit un seul état à un processus [Akl 1980, Kumar 1987, Steinberg 1990, Kalé 1990, Cung 1991], *parallélisation horizontale* [Roucairol 1990]. Elle peut être enfin une combinaison des deux [Finkel 1982, Hsu 1990, Powley 1991c] : certains processus ont chacun un seul état à traiter alors que d'autres ont chacun un sous-graphe d'états à explorer.

La stratégie d'exploration appartient en général à l'un des trois types suivants : *profondeur d'abord, largeur d'abord* ou *meilleur d'abord*.

Cette approche présente l'avantage d'être indépendante du domaine d'application, puisque les parcours de graphes d'états dépendent de l'algorithme utilisé et non de l'application.

Un des points cruciaux dans ce type de parallélisation est la communication entre les processus. Qu'elle soit faite par envois de messages en environnement distribué, ou par variables et structures de données partagées en environnement parallèle, elle doit éviter les situations de famine ou(et) de surcoût de recherche. Nous verrons dans la suite les diverses propositions faites, suivant le type d'environnement utilisé, qui visent à trouver un compromis entre un surcoût minimal de recherche et l'apparition d'un phénomène de famine.

1.3 Découpage de l'intervalle de recherche

L'idée d'utiliser le découpage de l'intervalle de recherche dans la parallélisation des parcours de graphes d'états, a été proposée pour la première fois par Baudet [Baudet 1978] pour l'algorithme $\alpha - \beta$. Cette approche est reprise par Powley et Korf [Powley 1989, Powley 1991c], puis Kalé et Saletore [Kalé 1990] et Li [Li 1991] pour l'algorithme IDA*.

Cette méthode consiste à partitionner un intervalle de recherche, auquel doit appartenir la valeur de l'état solution du problème, en plusieurs sous-intervalles. Chaque sous-intervalle est affecté à un processus. Chaque processus explore alors le graphe d'états sur son sous-intervalle. Les sous-intervalles étant plus petits que l'intervalle initial, ils permettent aux processus associés d'explorer moins d'états grâce aux élagages d'états effectués sur leur graphe. Ces états sont jugés, à l'aide d'une fonction d'évaluation, comme étant non prometteurs dans le sous-intervalle. Un des processus peut ainsi trouver plus rapidement une solution dans son sous-intervalle qu'un processus explorant tout l'intervalle initial.

Cette méthode est facilement applicable aux algorithmes utilisant des intervalles de recherche comme la fenêtre de recherche dans l'algorithme $\alpha - \beta$ et les seuils dans l'algorithme IDA*. Il reste à étudier dans quelle mesure elle peut être appliquée aux algorithmes de

type meilleur d'abord comme A*, SSS* [Stockman 1979] en IA et aux méthodes Branch-and-Bound avec stratégie d'exploration meilleur d'abord en RO.

Cette approche, employée seule, donne des accélérations limitées [Baudet 1978, Powley 1991c]. Mais combinée avec la décomposition des parcours de graphes d'états, elle pourrait permettre d'obtenir de meilleurs résultats dans l'avenir. Kalé et Saletore ont d'ores et déjà obtenu des résultats intéressants avec une approche mixte [Kalé 1990].

Chapitre 2

Approches MIMD du parcours de graphes d'états

Dans cette famille d'approches, nous avons regroupé les algorithmes dédiés aux machines multi-processeurs asynchrones à mémoire partagée ou non. Mais avant d'entrer dans l'analyse des algorithmes parallèles (section 2.3), voyons d'abord les différentes architectures (section 2.1) de machines et modes de programmation possibles pour les parcours de graphes d'états (section 2.2).

2.1 Architectures des machines MIMD utilisées

Dans cette classe de machines, les processeurs parallèles peuvent être centralisés sur une seule machine communicant à travers une mémoire partagée ou un réseau d'interconnexion interne, ou décentralisés sur plusieurs machines communicant à travers un réseau local de type Ethernet ou autres. Ces processeurs possèdent chacun une unité de contrôle indépendante et ils opèrent de façon asynchrone.

Nous pouvons essentiellement distinguer deux types de structures des processeurs :

1. les *arbres de processeurs* [Finkel 1982, Marsland 1985], les processeurs sont hiérarchisés : maîtres et/ou esclaves, et les communications s'effectuent uniquement entre un processeur maître et ses esclaves;

2. les *viviers de processeurs*, les processeurs ne sont pas hiérarchisés, tous les processeurs peuvent communiquer entre eux soit à travers une mémoire partagée [Akl 1980, Kumar 1987, Steinberg 1990, Kalé 1990, Cung 1991], soit à travers un réseau d'interconnexion [Baudet 1978, Rao 1987b, Li 1991, Powley 1991c].

Pour les parcours de graphes d'états, la tendance s'oriente actuellement vers les machines avec vivier de processeurs. Par conséquent, il n'y a plus de goulot d'étranglement au niveau des processeurs maîtres puisque tous les processeurs jouent le même rôle. De plus l'implantation des algorithmes y est plus souple, les contraintes architecturales posées par les arbres de processeurs n'ont pas à être prises en compte.

2.2 Deux modes de programmation parallèle

Les algorithmes d'exploration des graphes d'états s'expriment généralement de deux manières différentes suivant le mode de programmation sur lequel ils se fondent :

1. *structures de données* [Lecun 1991].
2. *structures de contrôle* [Schiper 1984, Florin 1991].

Le premier mode consiste à représenter explicitement les états à explorer par *une ou plusieurs structures de données* [Kumar 1987, Steinberg 1990, Li 1991, Cung 1991]. Des procédures de résolution sont associées aux structures de données et correspondent au parcours des graphes d'états.

L'exploration du graphe d'états est décrite par des procédures de résolution, et les états engendrés pendant l'exploration sont gardés dans les structures de données dynamiques. Ces dernières peuvent être partagées sur les machines disposant d'une mémoire partagée (cette mémoire partagée peut être virtuelle sur les machines à mémoire distribuée). Elles sont locales dans le cas où une machine à mémoire distribuée est employée.

Ce mode de programmation a l'avantage de nécessiter aucune modification particulière des langages de programmation classiques hormis les primitives de synchronisation et d'exclusion mutuelle.

Le second mode consiste à découper le graphe d'états en sous-graphes et à affecter chaque sous-graphe d'états à un ensemble de *processus de résolution* [Baudet 1978, Akl 1980, Finkel 1982, Marsland 1985]. Chaque processus évalue un état et engendre les états successeurs qui sont affectés à d'autres processus.

L'opération de génération des états est implicitement effectuée à travers les créations et les destructions de processus, car l'existence d'un état est liée à celle du processus associé.

Ce mode d'expression a l'avantage d'être plus naturel par rapport aux algorithmes de parcours de graphes d'états. Mais il nécessite la mise en œuvre d'une gestion complexe et automatique des processus par les systèmes d'exploitation pour affecter efficacement les processus aux processeurs. Cette gestion de processus se traduit éventuellement au niveau des langages de programmation par un ajout de structures de contrôle d'où son nom.

2.3 Les parallélisations à stratégie centralisée

La toile de fond des approches MIMD étant brossée, nous pouvons passer maintenant à l'analyse des parallélisations des algorithmes de parcours de graphes d'états. La première catégorie concerne les algorithmes parallèles à stratégie centralisée. Ils nécessitent l'utilisation d'une machine multi-processeurs à *mémoire partagée* (Cray-2, Encore Multimax, Sequent Balance et Symmetry, BBN Butterfly) pour la gestion des informations communes aux différents processus parallèles.

2.3.1 Une version parallèle centralisée de A* [Kumar 1987, Rao 1987c]

L'algorithme séquentiel A* [Nilsson 1980] utilise une file de priorité, appelée OUVERT, dans laquelle les états en attente d'exploration (génération et évaluation des états fils)

sont classés par ordre de mérite de leurs évaluations. A chaque itération de l'algorithme, le «meilleur» état est extrait de la liste et ses états fils sont alors engendrés et évalués. Le procédé est réitéré jusqu'à ce qu'un état final (s'il en existe un et c'est alors une solution optimale) soit extrait de la file OUVERT, ou jusqu'il n'y ait plus d'états à explorer dans la file. Le problème est alors résolu.

Principe de l'algorithme parallèle

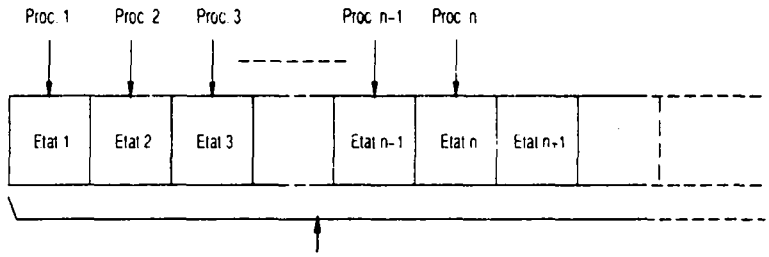

Figure 2.1 : File OUVERT d'états centralisée.

L'algorithme parallèle est fondé entièrement sur celui de A* en ce sens que n processus parallèles exécutent l'algorithme de base A*. Le nombre n est généralement le nombre de processeurs parallèles disponibles. Un processus est alors affecté à un processeur. Chaque processus va extraire de la file OUVERT le «meilleur» état à explorer. Les n processus explorent ainsi les n «meilleurs» états de la file OUVERT en parallèle (cf. Figure 2.1). Cette méthode est apparentée à une parallélisation horizontale.

Ce procédé est réitéré jusqu'à ce qu'un état final soit extrait de la file OUVERT ou, que les états dans la file de priorité soient épuisés et que tous les processus deviennent inactifs.

Structures de données

L'idée principale est d'utiliser une structure de données partagée, par tous les processus de résolution, associée à la file OUVERT de l'algorithme A*. L'accès à cette structure de données peut être exclusif ou concurrent.

L'accès exclusif ne permet cependant pas d'espérer de bonnes performances à cause des contentions d'accès, notamment lorsque le nombre de processus utilisés devient grand.

Dans ce cas, une structure à accès concurrent (une D-heap concurrente [Kumar 1987]) est intéressante. Les n processus parallèles peuvent alors travailler de manière concurrente sur la structure de données. L'accès à un état de la file de priorité ne bloque pas les accès à tous les autres états. Une étude comparative des files de priorité concurrentes peut être trouvée dans [Lecun 1991].

Problèmes posés par le parallélisme

L'avantage de cette parallélisation réside dans la simplicité de mise en œuvre. Mais elle pose quatre problèmes.

1. Les *contentions d'accès* malgré la structure partagée.

   Les performances des accélérations sont limitées à $T_{exp}/T_{accès}$ où T_{exp} est le temps de génération d'un état et $T_{accès}$ est le temps d'accès à un état.
   Les meilleures performances sont obtenues avec l'utilisation d'une file de priorité concurrente permettant aux processus parallèles d'accéder de façon concurrente aux états.

2. La *modularité de croissance*[1].

   L'accroissement du nombre de processeurs augmente d'une part les contentions d'accès à la structure partagée, et d'autre part les coûts de fabrication de la mémoire partagée des machines parallèles.

3. L'*anomalie d'accélération.*

   Du fait de l'exploration parallèle des n meilleurs états de la file OUVERT, il se peut qu'un état final soit trouvé en explorant moins d'états intermédiaires qu'en séquentiel, auquel cas une *accélération sur-linéaire* serait observée. Mais lorsqu'un état final nécessite plus d'exploration d'états qu'en séquentiel, on peut observer alors une *décélération*, voir même une anomalie *préjudiciable* [Roucairol 1990].

   La solution proposée par Kumar et al. est de terminer l'algorithme uniquement lorsque tous les états finaux sont explorés.

4. La *terminaison.*

   La condition d'arrêt utilisée par l'algorithme séquentiel n'est plus valide, car une file vide ne garantit pas qu'il n'existe plus d'état à explorer.

   Pour remédier à ce problème, il suffit de vérifier que tous les processus sont *inactifs* avant de terminer l'algorithme.

Résultats expérimentaux

Kumar et al. ont été les premiers à proposer une implantation parallèle de l'algorithme A* sur machine MIMD à mémoire partagée.

Ils l'ont appliquée aux problèmes du voyageur de commerce (15 et 25 villes) et de la couverture d'un graphe, et implantés sur une Butterfly avec 100 processeurs [Kumar 1987]. Une troisième application est le problème du taquin (4x4) implanté quant à lui sur une Sequent Balance 21000 avec 8 processeurs [Rao 1987c].

Pour le problème du voyageur de commerce et pour celui du taquin, les courbes d'accélération atteignent rapidement un palier lorsque l'on utilise une structure de données à accès exclusif : accélération de 40 sur la BBN Butterfly (100 processeurs) pour le voyageur de commerce avec 25 villes, et 3.1 sur la Sequent Balance (8 processeurs) pour le taquin. Avec une D-heap concurrente, les accélérations atteignent respectivement 96 et 7. L'utilisation d'une structure de données à accès concurrent améliore ainsi sensiblement les performances. Nous remarquons également que l'augmentation de la taille des problèmes repousse les paliers d'accélération, car la quantité de travail pouvant être effectuée en parallèle est plus importante.

[1] Scalability en anglais.

Cependant, sur le problème de la couverture d'un graphe, les performances sont annoncées comme médiocres sans être reportées. Cela est dû d'une part à l'utilisation d'une fonction d'évaluation f non discriminante provoquant ainsi l'exploration d'états non explorés en séquentiel. D'autre part, est posé le problème de la granularité : l'évaluation d'un état étant très simple, la quantité de travail affectée à chaque processus est petite, le temps de T_{exp} est faible devant $T_{accès}$, et le ratio $T_{exp}/T_{accès}$ est donc petit.

2.3.2 Une version parallèle centralisée de IDA* [Kalé 1990]

L'algorithme Iterative-Deepening A* (IDA*) [Korf 1985] a fait l'objet de plusieurs parallélisations ces dernières années, car il semble plus apte à être parallélisé que A*. Notamment son utilisation de seuils de recherche autorise un découpage en sous-intervalles.

L'algorithme IDA* séquentiel a été écrit à l'origine pour remédier à l'utilisation trop importante de mémoire de l'algorithme A* lors de la gestion de la file OUVERT. A l'instar de A*, IDA* n'utilise pas de file de priorité mais des *seuils de recherche*.

Le seuil de recherche initial est fixé à la valeur de l'état initial. Un nouveau seuil de recherche est fixé lorsque tous les états successeurs engendrés ont des valeurs supérieures à l'ancien seuil. La valeur de ce nouveau seuil est égale à la plus petite des valeurs des états ayant dépassé l'ancien seuil. Une nouvelle itération d'exploration commence alors avec ce nouveau seuil. L'algorithme procède jusqu'à ce qu'un état final soit atteint s'il existe.

La seconde différence avec l'algorithme A* est la stratégie d'exploration qui est du type *profondeur d'abord*. Cette stratégie est employée à chaque itération à l'intérieur d'un seuil de recherche par l'algorithme IDA*.

L'utilisation combinée des seuils de recherche et de la stratégie d'exploration profondeur d'abord permet d'«imiter» la stratégie d'exploration *meilleur d'abord*. L'avantage de cette combinaison se trouve dans le fait que la stratégie d'exploration profondeur d'abord autorise une complexité spatiale linéaire par rapport à la taille des problèmes traités.

Principe de l'algorithme parallèle

Le déroulement de l'algorithme IDA* parallèle est similaire à celui de l'algorithme séquentiel, sauf lors de l'exploration des états et de l'utilisation des seuils de recherche.

Entre chaque itération de l'algorithme IDA* parallèle, les états sont explorés en parallèle et de façon horizontale : un processus explore un état. Un *mécanisme de numérotation* des états permet aux processus parallèles d'effectuer une exploration en profondeur d'abord, en respectant ainsi l'ordre d'exploration de l'algorithme séquentiel. Les états les plus profonds et les plus à gauche dans le parcours arborescent du graphe sont numérotés avec la plus grande priorité; ils sont en conséquence les premiers à être explorés.

En complément à la décomposition du graphe de recherche, le découpage des seuils de recherche permet d'augmenter la quantité de travail à distribuer en parallèle.

Les processus parallèles sont autorisés à explorer les états d'un seuil supérieur si le nombre des états à explorer du seuil courant n'est plus suffisant et qu'un état final n'a pas été encore atteint (cf. Figure 2.2).

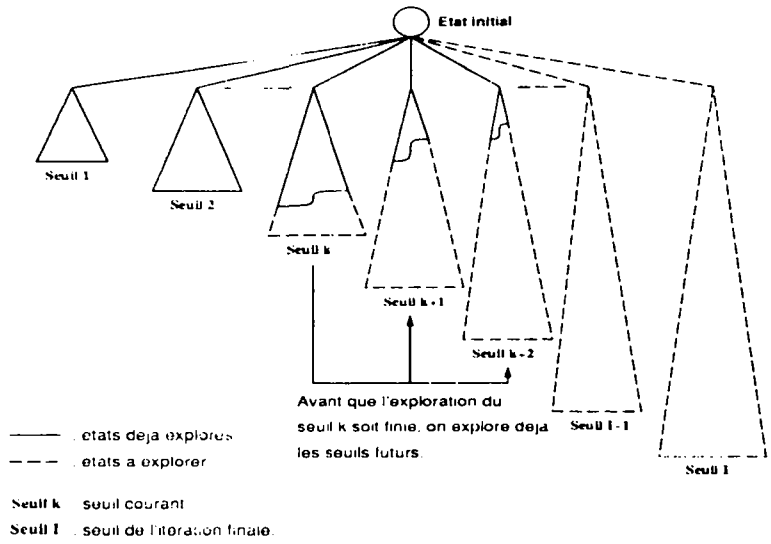

Figure 2.2 : Exploration des seuils par avance.

Structures de données

La gestion de la priorité des états peut être faite à l'aide d'une file de priorité à accès exclusif ou concurrente comme dans le cas de l'algorithme A* parallèle.

Problèmes posés par le parallélisme

Cet algorithme parallèle pose des problèmes de surcoûts de recherche, de terminaison et de détermination *a priori* des seuils de recherche.

1. Le *surcoût de recherche lié au parcours parallèle des graphes d'états.*

   Pour éviter un surcoût de recherche dû à l'exploration parallèle des états au début de l'arborescence de recherche, en absence d'informations utiles pour les élagages, les états fils d'un état courant ne sont pas engendrés immédiatement.

   Cette technique, appelée *delayed-release*, limite également la taille mémoire utilisée. Mais le parallélisme potentiel de l'algorithme se trouve réduit puisque les processus disponibles doivent attendre que ces états soient engendrés. Or si le nombre d'états fils engendrés est inférieur au nombre de processus disponibles, certains des processus peuvent rester inactifs.

   Il existe un compromis possible : si P processus sont disponibles, alors les P états les plus prioritaires (en accord avec la stratégie d'exploration profondeur d'abord) sont engendrés et explorés tant qu'un état final n'a pas été atteint. Un exemple d'exploration d'un arbre binaire avec quatre processus est donné sur la Figure 2.3.

   La combinaison du mécanisme de numérotation et de la technique de *delayed-release* permet d'avoir une exploration parallèle des graphes d'états très proche de l'exploration séquentielle. De ce fait les phénomènes d'anomalie d'accélération sont inexistants.

2. Le *surcoût de recherche lié au découpage des seuils de recherche.*

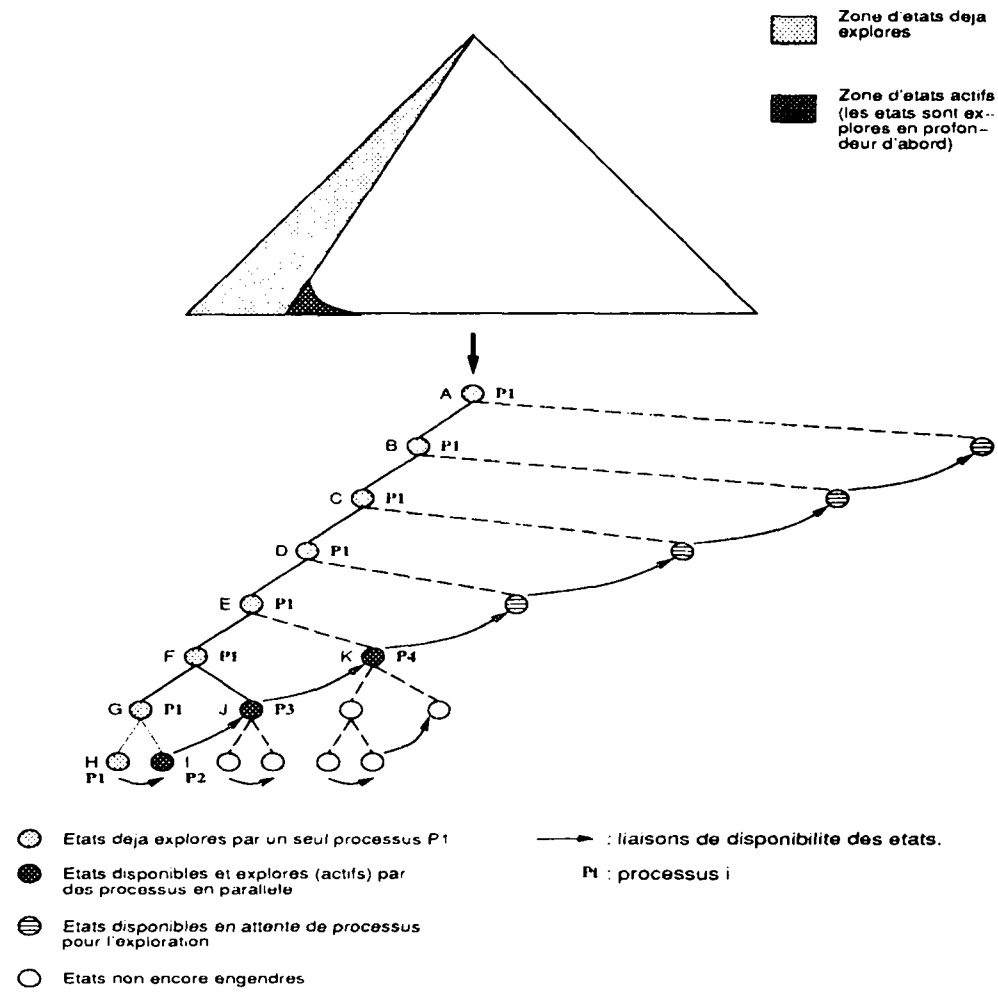

Figure 2.3 : Technique du *delayed-release*.

Une augmentation du surcoût de recherche lié au découpage des seuils de recherche apparaît si le nombre de seuils explorés par avance n'est pas limité. Des états vont alors être explorés dans des seuils de recherche qui ne seraient pas atteints en séquentiel.

3. La *terminaison liée à l'optimalité d'une solution trouvée*.

   Lorqu'un état final est atteint dans un seuil, afin de prouver son optimalité, il faut vérifier qu'aucun autre état final de seuil inférieur n'a été atteint. Si un autre état final dans un seuil inférieur a été trouvé, c'est sur cet autre état qu'il faut faire porter la vérification d'optimalité.

4. La *détermination* a priori *des seuils de recherche*.

   La technique d'exploration anticipée sur les seuils de recherche soulève une question majeure : comment peut-on connaître par avance les valeurs des seuils ?

   Pour un jeu comme le taquin, le calcul est simple. Nous savons que la valeur du seuil de l'itération $i + 1$ est égale à celle du seuil de l'itération i plus 2 ($Seuil_{i+1} =$

Seuil, + 2). Mais ces calculs de seuils sont moins évidents pour d'autres applications et la question de faisabilité demeure ouverte.

Résultats expérimentaux

L'implantation de Kalé et Saletore de cet algorithme semble être la meilleure à ce jour sur les machines MIMD à mémoire partagée. Ils l'ont appliqué au taquin 4x4 sur une Sequent Symmetry avec 20 processeurs.

Des accélérations presque linéaires sont obtenues pour une exploration de quatre seuils par avance : 18.5 (20 processeurs) pour un problème de petite taille (300 000 états explorés), et 18 (18 processeurs) pour un problème de grande taille (2 000 000 états explorés).

Il faut toute fois signaler que le parallélisme est géré par un environnement spécifique[2] leur permettant d'obtenir des bonnes performances avec des processus de granularité moyenne.

2.4 Les parallélisations à stratégie distribuée

Les algorithmes à stratégie distribuée induisent des processus communicant entre eux par envois de messages. Des machines allant des multi-processeurs à mémoire partagée aux stations de travail interconnectées par un réseau local sont donc utilisables.

L'utilisation d'une mémoire distribuée a l'avantage de conduire à une extensibilité plus aisée au niveau des architectures. De ce fait, le coût de fabrication est moins élevé.

Au niveau des algorithmes, l'absence de mémoire partagée entraîne la disparition de structures de données partagées. Les contentions d'accès à ces dernières ne se posent donc plus. Cependant, les communications inter-processus deviennent cruciales. Des surcoûts importants de recherche et un phénomène de famine peuvent être provoqués par la non communication de données globales.

2.4.1 Une version parallèle distribuée de A* [Kumar 1987, Rao 1987c]

Nous avons vu que la file OUVERT de A* est implantée en mémoire partagée avec une structure de données à accès concurrent (section 2.3.1). En absence totale de mémoire partagée, il est possible de faire gérer cette file OUVERT par un processus spécialisé, mais la fréquence élevée des communications entre ce processus spécialisé et les autres processus risque alors de faire écrouler les performances souhaitées. La seule solution envisageable est le découpage de la file OUVERT en plusieurs files OUVERT *locales* à chaque processus.

Principe de l'algorithme distribué

Nous supposons qu'un seul processus est affecté à un et un seul processeur. Chaque processus exécute l'algorithme séquentiel A* avec une file OUVERT locale aux processus. Le meilleur état de chaque file sera donc toujours exploré.

[2] «Chare-kernel».

Structures de données

Dans cet algorithme, les structures de données sont simples. Chaque file OUVERT locale est implantée sous forme de file de priorité. L'accès concurrent aux files de priorité n'est pas utile puisque chaque processus accède uniquement à sa propre file de priorité.

Problèmes posés par le parallèlisme

Nous retrouvons pour cet algorithme les deux problèmes posés par la version centralisée : anomalie d'accélération et *terminaison*. Les solutions sont aussi les mêmes.

Nous avons cependant en plus le problème classique des algorithmes distribués : l'*équilibrage des charges* du travail. Un mauvais équilibrage des charges provoquera à la fois des surcoûts de recherche et des phénomènes de famine. Un «bon» équilibrage des charges passe nécessairement par une bonne stratégie de communication entre les processus. Or le choix d'une stratégie dépend du coût des communications. Cela nous pousse alors à connaître l'organisation du support des communications: *architecture du réseau.*

Définition 1 *Diamètre d'un graphe.*
Soit $d_{i,j}$ la longueur d'un chemin entre deux sommets quelconques i et j d'un graphe. Alors le diamètre D d'un graphe est

$$D = \max(\min_{i,j} d_{i,j}).$$

En général, plus le *diamètre* d'un réseau est faible, moins le coût de communication est élevé. Par conséquent, lorsque le diamètre d'un réseau est petit, le passage d'informations d'un processeur vers les autres nécessite moins de processeurs intermédiaires.

Nous analysons trois stratégies de communication pour les réseaux de petit, moyen et de grand diamètre.

Communication par tableau noir

La communication par tableau noir consiste à garder des états évalués dans un endroit, matérialisé par soit un processus spécialisé, soit un espace en mémoire partagée. La politique de communication est la suivante :

- dès qu'un processus a dans sa file locale des états de valeurs inférieures à la plus petite valeur des états du tableau diminué d'un seuil ϵ (fixé à l'avance), il va transférer quelques états du tableau noir vers sa file locale;
- dès qu'un processus a dans sa file locale des états de valeurs supérieures à la plus grande valeur des états du tableau augmenté d'un seuil ϵ, il va transférer quelques états de sa file locale vers le tableau noir.

Ainsi, un état solution optimale est nécessairement exploré.

Le seuil ϵ détermine la fréquence de transfert des états entre le tableau noir et les processus. Si ϵ est petit, il y aura beaucoup de communications, mais moins de surcoûts de recherche et de famine. Il importe donc d'ajuster ce seuil.

L'inconvénient majeur de cette stratégie est l'apparition de contentions d'accès au tableau noir qui limite fortement la modularité de croissance de l'algorithme.

Communication aléatoire

Les processus communiquent entre eux de façon aléatoire : de temps à autre un processus envoie un état à explorer à un processus tiré au sort. Ce type de communication est bien adapté aux multi-processeurs organisés en réseau de petit diamètre. En effet, les communications sont relativement nombreuses et les coûts peuvent être élevés si le diamètre est grand.

Communication en anneau

Les processus communiquent uniquement avec leurs deux voisins. Périodiquement, un processus envoie un état nouvellement engendré à son voisin (de droite ou de gauche, fixé une fois pour toute). Ce type de communication est bien adapté au réseau de processeurs de grand diamètre.

Résultats expérimentaux

Ces trois stratégies de communication pour l'algorithme distribué de A* ont été proposées par Kumar et al., et elles ont été testées sur les problèmes du voyageur de commerce et de la couverture d'un graphe. L'algorithme est implanté sur une BBN Butterfly avec 100 processeurs.

Pour le voyageur de commerce, les meilleures accélérations sont obtenues avec la communication par tableau noir 69 (83 processeurs). Avec la communication aléatoire, on obtient une accélération de 23 (60 processeurs) et pour la communication en anneau 4 (au delà de 10 processeurs).

De meilleures performances sont obtenues avec le problème de la couverture d'un graphe : 56.7 (60 processeurs) pour la communication par tableau noir, 49 (60 processeurs) pour la communication aléatoire, et 26 (60 processeurs) pour la communication en anneau. Le fait que la fonction d'évaluation soit non discriminante justifie l'amélioration des accélérations. Beaucoup d'états ayant la même valeur sont alors engendrés par les processus et la communication de ces états entre processus devient inutile. Les stratégies distribuées en général sont de ce fait favorisées par rapport à celle centralisée donnée à la section 2.3.1 sur le même problème traité.

2.4.2 Trois versions parallèles distribuées de IDA*

Plusieurs équipes [Rao 1987b, Powley 1989, Li 1991] ont proposé ces dernières années des parallélisations de IDA* sur des machines multiprocesseurs à mémoire distribuée, allant des machines comme BBN Butterfly et Hypercube aux réseaux d'ordinateurs.

L'algorithme IDA* offre un parallélisme potentiel supérieur à celui de A* grâce à l'utilisation de seuils de recherche. Chaque seuil de recherche est exploré par au moins un processus.

Certains des problèmes déjà analysés pour la version parallèle centralisée de cet algorithme (section 2.3.2) se posent de façon cruciale pour les versions parallèles distribuées. Rappelons ici les principaux problèmes posés :

1. la *terminaison liée à l'optimalité de la solution trouvée*,

2. l'ignorance *a priori* de seuils de recherche, sauf pour le taquin;

3. la *communication inter-processus* pour éviter à la fois la famine et le surcoût de recherche.

Le dépassement anticipé des seuils de recherche permet de faire un travail spéculatif et d'éviter les situations de famines à la fin ou au début d'une itération. Cependant ce travail spéculatif peut provoquer l'obtention de solutions non optimales. Il faut vérifier alors que toutes les solutions des itérations antérieures à l'itération de la solution trouvée sont moins bonnes pour s'assurer de l'optimalité d'une solution. S'abstenir de cette recherche permet toutefois de proposer des algorithmes approchés performants [Powley 1989, Powley 1991c, Li 1991].

La nécessité de connaître les seuils de recherche est propre à l'algorithme IDA* parallèle. Elle est déjà discutée dans la version parallèle centralisée (section 2.3.2).

Quant au problème de communication inter-processus, il n'existe pas encore de solution générale. Un consensus semble pourtant se dégager sur la méthode à adopter pour la décomposition des graphes d'états. La *parallèlisation horizontale* nécessitant trop de communications inter-processus est abandonnée au profit de la *parallèlisation verticale*. Chaque processus s'occupe désormais d'un sous-graphe d'états. Ainsi la communication de sous-graphes d'états d'un processus P_i vers un autre P_j aura lieu uniquement lorsque le processus P_j a fini d'explorer le sous-graphe qui lui est attribué initialement.

Nous allons analyser dans les sections suivantes les différentes solutions apportées à ces problèmes.

Algorithme Parallel-IDA* [Rao 1987b, Rao 1987a]

Rao et al. ont proposé en 1987 la première parallélisation de *IDA**. Ils utilisent une décomposition verticale de l'arbre de recherche à chaque itération de l'algorithme. Mais leur algorithme n'effectue pas de dépassement anticipé des seuils de recherche des itérations futures.

Principe de l'algorithme Parallel-IDA*

Chaque processus exécute IDA* séquentiel sur un sous-graphe qui lui est attribué initialement. Les processus se synchronisent sur les seuils de recherche à chaque itération. Un seuil global de recherche est donné à l'ensemble des processus au début de chaque itération.

Structures de données

La stratégie de recherche en profondeur d'abord de IDA* autorise l'utilisation d'une simple file de type *dernier entré premier sorti*[3], communément appelée *pile*, pour chaque processus.

Dans la décomposition verticale du graphe de recherche, chaque processus gère une pile locale d'états contenant les états de son sous-graphe. Les états engendrés sont mis

[3] Last In First Out (LIFO) en anglais

dans la pile. Ainsi les états les plus profonds dans les arborescences de recherche des sous-graphes sont toujours en haut des piles locales correspondantes. La stratégie de recherche *profondeur d'abord* est alors respectée.

Problèmes posés par le parallèlisme

L'algorithme distribué souffre principalement de trois défauts liés à :

1. la *communication inter-processus* des états à explorer,
2. l'existence du *phénomène de famine* aux synchronisations aux seuils de recherche entre deux itérations,
3. l'existence d'*anomalies d'accélération* : phénomène d'*accélération sur-linéaire* et de *décélération*.

La politique de partage des états peut aboutir à un déséquilibre des charges de travail, si les tailles des sous-graphes sont différentes, et à un manque de communications entre processus, provoquant ainsi des situations de surcout de recherche et de famine.

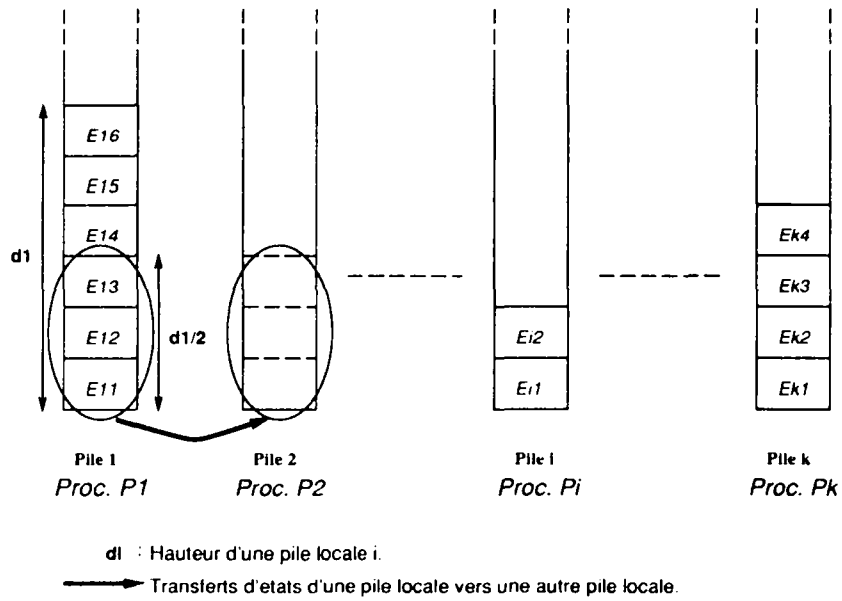

Figure 2.4 : Communication des états entre piles locales.

La stratégie de communication choisie est la suivante. Appelons $Pile_i$ la pile locale de hauteur d_i d'un processus P_i. La pile $Pile_i$ est représentée du *sommet* vers la *base*. Les états en haut de l'arborescence de recherche du sous-graphe du processus P_i sont à la base de la pile $Pile_i$. Le nombre d_i représente la hauteur à un instant donné de la pile $Pile_i$. Le processus P_i communique uniquement les états se trouvant à des hauteurs inférieures à $d_i/2$ (cf. Figure 2.4).

Cette stratégie a deux raisons d'être. D'une part, elle permet d'éviter des contentions d'accès trop importantes aux piles locales dues aux communications inter-processus, puisque les états se trouvant à une hauteur supérieur à $d_i/2$ d'une pile locale $Pile_i$ ne sont pas communiqués à d'autres processus. D'autre part, en communiquant les états à la base d'une pile, on communique en réalité les états du haut de l'arborescence de recherche du sous-graphe correspondant. Or les états du haut d'une arborescence de recherche sont générateurs potentiels d'une quantité de travail importante puisqu'ils nécessitent généralement une exploration d'un grand nombre d'états fils. Ainsi la communication des états du haut d'une arborescence de recherche permet à peu de coût le transfert d'une quantité de travail importante d'un processus vers un autre.

Cependant cette stratégie de communication demande aux processus sans travail (inactifs) d'aller vérifier *toutes* les piles locales des autres processus afin de trouver des états à explorer. Cette recherche coûte très cher en communication.

Le deuxième défaut est qu'il n'existe pas de recherche spéculative utilisant la technique de dépassement anticipé des seuils de recherche. Ainsi à la fin d'une itération d'un seuil et au début de l'itération du seuil suivant, il peut y avoir un phénomène de famine.

Les expériences menées avec le taquin 4x4 ont montré l'existence d'accélérations surlinéaires et de décélérations. Elles sont liées au non déterminisme des explorations parallèles à la dernière itération de l'algorithme distribué. Pour résoudre ce problème, une modification de la condition d'arrêt de l'algorithme est nécessaire. On s'arrête seulement lorsque toutes les solutions optimales ont été trouvées.

Résultats expérimentaux

L'implantation de cet algorithme a été effectuée sur une Sequent Balance 21000 par Rao et al.. Des accélérations presque linéaires ont été obtenues : 27.6 pour 30 processeurs. Cependant, sur des machines plus faiblement couplées que la Sequent Balance 21000, nous pouvons raisonnablement nous attendre à des performances moindres à cause des coûts de communication plus élevés.

Algorithme Parallel Window Search [Powley 1989, Powley 1990, Powley 1991c]

L'algorithme *Parallel Window Search* (PWS) est proposé par l'équipe même du concepteur de IDA* séquentiel [Korf 1985]. Il est fondé sur deux idées principales.

La première est le découpage des seuils de recherche, un ou plusieurs processus sont affectés à chaque seuil de recherche correspondant à une itération de IDA* (cf. Figure 2.5). La seconde consiste en un ordonnancement d'un certain nombre d'états suivant les valeurs fournies par la fonction d'évaluation. Les auteurs ont prouvé que ces deux idées sont complémentaires et contribuent à donner de bonnes performances à PWS. Cependant ils n'ont pas utilisé la décomposition du graphe de recherche dans leur étude bien qu'ils l'aient suggérée.

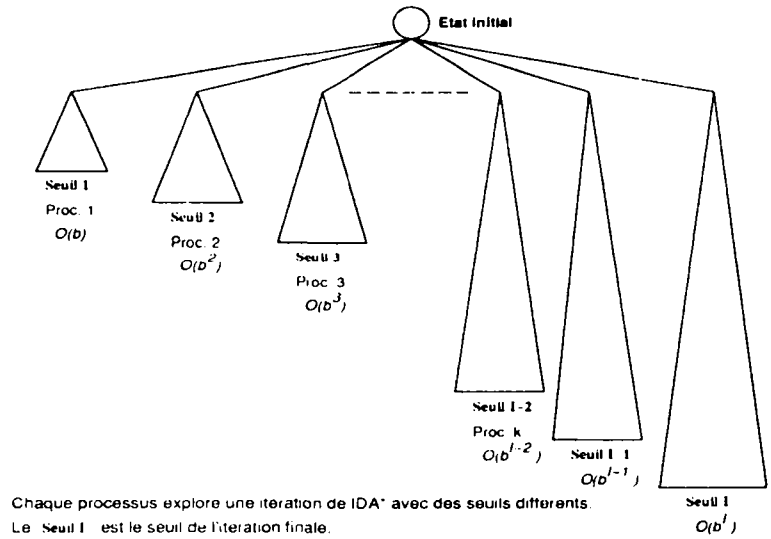

Figure 2.5 : Affectation de processus aux seuils de recherche.

Découpage des seuils de recherche

Powley et al. sont les premiers à proposer, en 1989, une parallélisation des seuils de recherche de IDA*. Cette idée est reprise ensuite par Kalé et Saletore [Kalé 1990] dans une version parallèle de IDA* (section 2.3.2).

En théorie, nous avons les propriétés suivantes sur l'amélioration qu'apporte le découpage des seuils de recherche.

Propriété 1 *L'affectation d'un processus à chaque seuil de recherche de IDA* permet d'espérer un temps de résolution parallèle majoré par*

$$ab^I(\frac{b}{b-1})$$

où

- b *est le facteur de branchement* heuristique *[Nilsson 1980],*
- I *la profondeur de l'état solution dans l'arbre de recherche,*
- a $(0 < a \leq 1)$ *est la fraction d'états à explorer afin de trouver un solution optimale dans l'itération finale.*

Puisque la somme

$$b^I + b^{I-1} + \cdots + b^2 + b,$$

représentant le nombre total d'états explorés dans le seuil final, est majorée par la somme

$$b^I + b^{I-1} + \cdots + b^2 + b + 1 + 1/b + 1/b^2 + \cdots$$

convergeant pour $b > 1$ vers

$$b^l(\frac{b}{b-1}).$$

Ce temps de résolution est en fait le temps que met le processus affecté au seuil final pour trouver une solution. C'est donc également le temps que met l'agorithme IDA* pour trouver une solution au seuil final.

En nous rappelant que l'algorithme IDA* a, pour tous les seuils antérieurs au seuil final I, un temps de résolution égal à

$$b^{l-1} + 2b^{l-2} + \cdots + (I-2)b^2 + (I-1)b$$

qui peut être majoré par

$$b^{l-1} + 2b^{l-2} + \cdots + (I-2)b^2 + (I-1)b + I + (I+1)/b + (I+2)/b^2 + \cdots.$$

Cette série converge pour $b > 1$ vers

$$b^{l-1}(\frac{b}{b-1})^2.$$

Nous obtenons alors un temps total de résolution pour IDA*, tous les seuils inclus, égal à

$$b^{l-1}(\frac{b}{b-1})^2 + ab^l(\frac{b}{b-1}).$$

Nous en déduisons le corollaire suivant.

Corollaire 1 *L'accélération espérée, en utilisant le découpage des seuils de recherche, de PWS par rapport à IDA* est alors de*

$$1 + \frac{1}{a(b-1)}$$

où

- *b est le facteur de branchement* heuristique.
- *a $(0 < a \leq 1)$ est la fraction d'états à explorer afin de trouver un solution optimale dans l'itération finale.*

Cette formule montre clairement que les gains apportés par le découpage des seuils de recherche sont fortement limités par le facteur a de l'itération finale, surtout lorsque le facteur a est grand.

Ordonnancement des états

Pour éliminer cette limite provoquée par le facteur a, deux solutions sont possibles. La première consiste à utiliser plusieurs processus, comme Kalé et Saletore, pour l'exploration de chaque itération. Ainsi le temps de recherche de l'itération finale peut être réduit.

La seconde, proposée par Powley et al., ordonnance les états suivant leurs valeurs de la *fonction d'évaluation* f. En cas d'égalité des évaluations de f, les états sont classés par

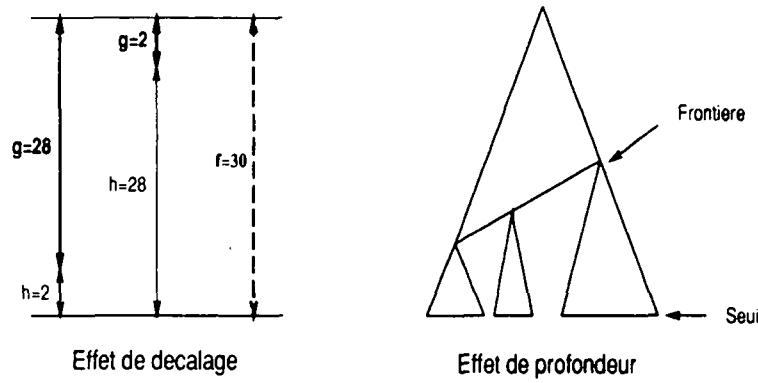

Figure 2.6 : Les deux effets dûs à l'ordonnancement des états.

ordre croissant des valeurs de la *fonction heuristique* h. Cet ordonnancement est effectué à chaque itération de l'algorithme.

Cet ordonnancement des états provoque deux types d'effets : *décalage* et *profondeur* (cf. Figure 2.6).

Ces deux effets contribuent tous deux à diminuer le facteur a et à améliorer donc l'accélération théorique de PWS.

L'effet de décalage est provoqué par le fait qu'en explorant les états de plus petites heuristiques d'abord, en cas d'égalité des évaluations de f, nous espérons trouver plus rapidement une solution optimale. En effet la fonction heuristique h sous-estiment toujours les valeurs des chemins réels menant aux solutions optimales, puisqu'elle est toujours minorante (ou admissible). Par conséquent, plus les valeurs heuristiques sont petites, plus les valeurs de g correspondantes sont grandes et donc la longueur du chemin déjà parcourue également. Cela signifie que nous sommes trés proche d'une solution optimale.

Aussi l'ordonnancement proposé produit un second effet appelé *profondeur* qui est en fait une conséquence de l'effet de décalage. Comme choisir d'explorer d'abord les états d'heuristiques faibles, en cas d'égalité des évaluations de f, permet d'avoir plus de précision sur les valeurs réelles des chemins menant aux solutions optimales. Nous avons donc un sous-arbre de recherche de taille moins grande à explorer pour trouver une solution optimale si elle existe. Par la même occasion, nous pouvons éviter l'exploration inutile de certains états et diminuer le facteur a.

Cependant un ordonnancement global des états nous obligerait à garder en mémoire tous les états explorés dans une itération antérieure. De ce fait, la complexité spatiale linéaire [Korf 1985] de IDA* redevient exponentielle et nous perdons l'intérêt essentiel de IDA* sur A*.

Powley et al. choisissent un compromis entre l'occupation spatiale et l'ordonnancement global des états. Au début de l'exploration d'un graphe d'états, l'état initial est développé avec l'algorithme A* ainsi que les états successeurs jusqu'à l'obtention d'un ensemble de 100 à 1000 meilleurs *états-frontière*. Ces états-frontières sont explorés jusqu'au seuil courant, puis ils sont classés dans l'ordre décroissant de la longueur des chemins explorés (ce qui correspond à classer les états par valeur croissante de h) pour l'itération suivante avec un seuil supérieur. Ainsi l'occupation mémoire reste limitée, mais l'ordonnancement (partiel) est uniquement sur les états-frontières et non sur tous les états au seuil d'une

itération (cf. Figure 2.7).

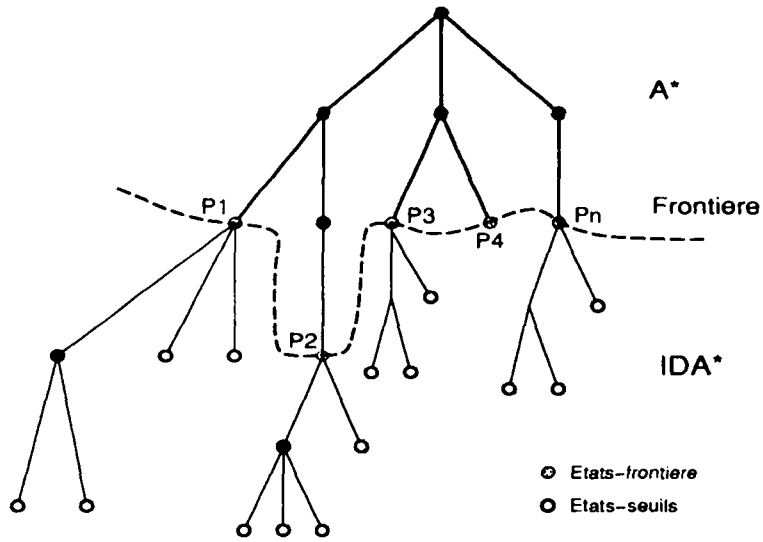

Figure 2.7 : Etats-frontières et états-seuils.

Sur un plan théorique, nous avons la propriété suivante sur l'apport de l'ordonnancement des états.

Propriété 2 *L'amélioration maximale apportée à l'algorithme IDA* par un ordonnancement global des états est de l'ordre de b en terme d'états explorés, où b est le facteur de branchement heuristique de l'arborescence de recherche du graphe d'états.*

En effet dans *le pire des cas*, la solution se trouve, au seuil final, à droite de l'arborescence du graphe de recherche. L'algorithme IDA* doit explorer tous les états du seuil final, ce qui donne comme temps de résolution :

$$b^I + 2b^{I-1} + \cdots + (I-2)b^3 + (I-1)b^2 + Ib.$$

Or dans *le meilleur des cas*, la solution se trouve, au seuil final, à gauche de l'arborescence du graphe de recherche. Avec la stratégie de recherche en profondeur d'abord de IDA*, la dernière itération nécessite uniquement l'exploration de d états. Le nombre d représente la longueur du chemin de l'état initial vers l'état final. Nous obtenons alors un temps de résolution de

$$b^{I-1} + 2b^{I-2} + \cdots + (I-2)b^2 + (I-1)b + d.$$

Le rapport de ces deux temps en approximant d par I nous donne le facteur d'amélioration escompté b.

Empiriquement en terme de temps, les performances par rapport à IDA* sans ordonnancement donne une amélioration de 1,1 à 1,83. Il faut aussi noter que cette amélioration est portée essentiellement sur l'itération finale de IDA*.

Principe de l'algorithme PWS

Cet algorithme combine les avantages complémentaires du découpage des seuils de recherche et de l'ordonnancement des états.

Supposons que nous ayons P processus disponibles. Dans une première phase d'initialisation, l'algorithme PWS affecte respectivement les P premiers seuils de recherche aux P processus. Puis un ensemble identique de 100 à 1000 états-frontière est engendré suivant l'algorithme A* par chaque processus. Si une solution est trouvée dans cette phase alors l'algorithme se termine, sinon chaque ensemble d'états-frontière est exploré suivant l'algorithme IDA* par le processus correspondant jusqu'au seuil de recherche initialement affecté.

Lorsqu'un processus a exploré son seuil de recherche, il va réordonner son ensemble d'états-frontière dans l'ordre croissant des valeurs de h. Ce processus diffuse alors les informations d'ordonnancement aux autres processus. Ces informations concernent :

1. la valeur h au seuil de recherche de chaque état-frontière et le chemin associé,
2. la valeur du seuil de recherche du processus.

Puis un nouveau seuil de recherche n'ayant pas encore été affecté, le sera sur ce processus.

A la réception des informations d'ordonnancement, un processus arrête sa recherche sur son ensemble d'états-frontière et réordonne les états-frontière qui n'ont pas encore été explorés. Seules les informations d'ordonnancement provenant du plus grand seuil sont prises en compte.

L'arrêt de l'algorithme se fait à la première solution optimale trouvée et vérifiée.

Structures de données utilisées

En ce qui concerne les états-frontière, un tableau local à chaque processus suffit pour stocker l'ensemble des états-frontière correspondant trié, et une file de priorité locale est nécessaire pour gérer le tri de cet ensemble.

Pour l'exploration en profondeur d'abord des états-frontière, nous pouvons utiliser une simple pile locale pour chaque processus.

Problèmes posés par le parallèlisme

Quatre problèmes sont soulevés par l'algorithme PWS :

1. la *terminaison liée à l'optimalité de la solution trouvée*,
2. l'ignorance *a priori* des seuils de recherche,
3. le *surcoût de recherche lié au découpage des seuils de recherche*,
4. l'ignorance *a priori* du nombre de seuils total pour la résolution d'un problème.

Les trois premiers problèmes sont déjà discutés (section 2.3.2). En ce qui concerne le surcoût, il faut toutefois ajouter un surcoût lié aux tris *partiels* des ensembles des états-frontière à la réception des informations d'ordonnancement par des processus. Les auteurs prétendent cependant que ce surcoût est faible.

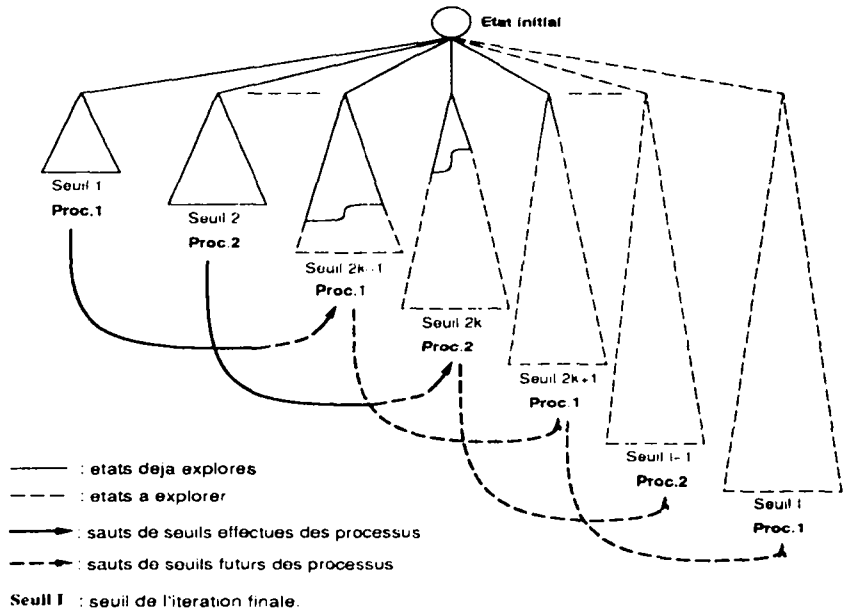

Figure 2.8 : Sauts de processus d'itération en itération.

On ignore *a priori* le nombre de seuils total nécessaire à la résolution du problème. La complexité de PWS ne peut être calculée qu'en fonction de ce nombre.

Propriété 3 *La complexité de l'algorithme PWS, avec $I/2$ processus et un ordonnancement global au seuil $I/2$, est en*

$$O(b^{I/2})$$

où

- *b est le facteur de branchement heuristique.*
- *I le nombre de seuils que l'algorithme IDA* explorerait pour trouver un premier état final (ou une première solution).*

De façon générale, si nous disposons de N processus alors un processus J explore les itérations J, $N + J$, $2N + J$, $3N + J$ .... La Figure 2.8 illustre un exemple de sauts de deux processus de seuil en seuil.

En conséquence, avec $I/2$ processus, le processus $I/2$ explore l'itération $I/2$ et une fois l'exploration terminée, il ira directement à l'itération finale I ($I = I/2 + I/2$). Sous l'hypothèse que l'ordonnancement est effectué globalement au seuil $I/2$, nous avons alors une complexité temporelle égale à $O(b^{I/2} + b^{I/2})$ soit $O(b^{I/2})$.

Cependant, cette complexité temporelle ne peut être obtenue en pratique et ceci pour deux raisons. La première est liée au facteur I qui ne peut être connu a priori, la deuxième

à l'ordonnancement global des états à l'itération $I/2$. Un ordonnancement global demanderait une complexité spatiale trop importante et une complexité temporelle supplémentaire.

Résultats expérimentaux

L'algorithme PWS a été implanté sur quatre systèmes différents : un DEC VAX monoprocesseur en temps partagé, un réseau de 20 IBM-PC, un réseau de HP 9000/320 et 350 stations de travail, et un Hypercube d'Intel.

Curieusement, les auteurs n'ont pas présenté les résultats en terme d'accélération mais en terme de réduction exponentielle du nombre d'états engendrés.

Définition 2 *Facteur de réduction expotentielle.*
Soient N_A, N_B le nombre d'états engendrés respectivement par l'algorithme A et B. Le facteur de réduction exponentielle c *est tel que*

$$(N_A)^c = N_B.$$

En prenant l'algorithme IDA* pour A et l'algorithme PWS pour B, trois *facteurs* entraînant une réduction exponentielle ont été présentés :

1. la recherche d'une première solution (pas nécessairement optimale),
2. celle d'une solution optimale,
3. celle d'une solution optimale avec la preuve de son optimalité.

Première solution	Solution optimale	Sol. opt. vérifiée
0.73	0.83	0.96

Tableau 2.1 : Facteurs de réduction expotentielle.

Le tableau 2.1 résume les facteurs c calculés sur les 44 problèmes de taquin 4x4 les plus faciles proposés par Korf [Korf 1985], avec 7 processeurs.

Écart de la solution optimale	0	2	4	6	8	10	12
Nombre de problèmes	1	8	24	37	22	6	2

Tableau 2.2 : Qualité des premières solutions.

La *qualité* des premières solutions par rapport aux solutions optimales, exprimée en nombre de mouvements, est également importante pour apprécier les performances de l'algorithme. Le tableau 2.2 résume les écarts entre les premières solutions et les solutions optimales. Ces résultats sont obtenus sur les 100 problèmes de taquin 4x4 avec 7 processeurs. Les premières solutions sont rapportées comme étant en moyenne à 11% des solutions optimales.

Malheureusement aucune des deux mesures de performance ne nous permet d'avoir des indications sur les accélérations réelles de cet algorithme et de le comparer aux autres.

Algorithme Imprecise-IDA* [Li 1991]

L'algorithme *Imprecise-IDA** est, comme son nom l'indique, uniquement un algorithme parallèle approché de l'algorithme IDA*. Il utilise la décomposition verticale des graphes de recherche et le découpage des seuils de recherche.

Principe de l'algorithme Imprecise-IDA*

Comme l'algorithme Parallel-IDA* (section 2.4.2), chaque processus explore un sous-graphe de recherche dans sa pile locale. Mais contrairement à Parallel-IDA*, un processus peut communiquer tous les états de sa pile locale dès que celle-ci a une hauteur supérieure ou égale à deux.

L'originalité d'Imprecise-IDA* se situe dans sa manière d'explorer les seuils de recherche. Li propose de rechercher une solution non nécessairement optimale dans l'intervalle initial $[a, b]$ qui se rétrécit au cours de l'exécution de l'algorithme, où a est la valeur de l'état *racine* fournie par la fonction d'évaluation, soit $f(racine)$, et b une borne supérieure fixée. L'algorithme Imprecise-IDA* se termine lorsque la différence entre les deux bornes de l'intervalle est inférieure à une valeur ϵ fixée.

L'application test utilisée, en l'occurence le problème d'affectation quadratique (QAP) et non le taquin 4x4, ne permet pas de connaître a priori les valeurs de seuils de recherche et par conséquent d'utiliser le découpage des seuils de recherche dans la parallélisation de l'algorithme IDA*. Cependant, il est tout de même possible de franchir les seuils des itérations. L'algorithme Imprecise-IDA* recherche, à chaque itération $i+1$, une solution avec comme seuil de recherche le minimum des valeurs des états de l'itération i ($\min F_i$ comme IDA*) augmenté d'une certaine valeur Δ.

A chaque itération $i+1$, si $[a_i, b_i]$ est l'intervalle obtenu après i itérations de Imprecise-IDA*, la valeur du seuil à l'itération $i+1$ est déterminée par l'algorithme suivant :

- si une solution est trouvée à l'itération i alors b_{i+1} est la valeur de cette solution et
  - si a_i a été modifiée à l'itération i alors

$$Seuil_{i+1} = \min F_i + 0,5 * (b_{i+1} - a_i) \quad \text{avec} \quad \Delta = 0,5 * (b_{i+1} - a_i),$$

  - sinon

$$Seuil_{i+1} = \min F_i + 0,9 * (b_{i+1} - a_i) \quad \text{avec} \quad \Delta = 0,9 * (b_{i+1} - a_i);$$

- sinon

$$a_{i+1} = \min F_i$$

  - si b_i a été modifiée à l'itération i alors

$$Seuil_{i+1} = \min F_i + 0,5 * (b_i - a_{i+1}) \quad \text{avec} \quad \Delta = 0,5 * (b_i - a_{i+1}),$$

  - sinon

$$Seuil_{i+1} = \min F_i + 0,1 * (b_i - a_{i+1}) \quad \text{avec} \quad \Delta = 0,1 * (b_i - a_{i+1}).$$

Les facteurs 0.9 et 0.1 ont pour rôle d'accélérer le resserrement de l'intervalle $[a, b]$ respectivement en s'*éloignant* de la borne a si elle n'a pas été mise à jour à l'itération $i-1$ et si une solution a été trouvée à l'itération i, et en se *rapprochant* de a si b n'a pas été mise à jour à l'itération $i-1$ et si aucune solution n'a été trouvée à l'itération i. Le facteur 0.5 resserre quant à lui l'intervalle $[a, b]$ de moitié à chaque itération.

L'implantation a été faite sur un anneau de processeurs esclaves supervisé par un processeur maître pour les communications inter-esclaves. A chaque processeur esclave a été affecté un seul processus.

Structures de données utilisées

La stratégie de recherche en profondeur d'abord et l'utilisation de la décomposition des graphes de recherche imposent la gestion d'une simple pile locale à chaque processus.

Problèmes posés par le parallèlisme

L'utilisation d'un anneau de processeurs esclaves supervisés par un processeur maître peut a priori poser un problème de contention d'accès au niveau du processeur maître.

La stratégie de communication entre les processus esclaves proposée par Li s'applique de la façon suivante. A tout instant le processus maître connaît, par communication maître-esclave, les processus esclaves qui ont encore des états à explorer. Lorsqu'un processus esclave n'a plus d'états à explorer dans sa pile locale, il va faire une demande auprès du processus maître qui lui indique en retour les processus esclaves possédant encore du travail afin qu'il puisse en acquérir.

Cette politique de communication permet d'éviter à un processus inactif de vérifier tous les autres processus pour trouver du travail. La fréquence des communications entre processus est ainsi réduite. L'auteur prétend que cette configuration en anneau de processus esclaves permet des coûts de communication réduits.

Autant cette solution nous semble pouvoir obtenir de bonnes performances pour un nombre de processeurs restreint à une dizaine, autant elle nous semble inopérante dans une perpective de mise en oeuvre de parallèlisme massif.

Résultats expérimentaux

Les résultats présentent des temps de résolution meilleurs que ceux de Rao et al. pour 2, 4 et 8 transputers.

Les nombres d'états explorés sont également moindre en comparaison à ceux de l'algorithme IDA* séquentiel. Cependant nous observons des accélérations sur-linéaires pour les instances de taille 10x10 et 10x20 des problèmes testés.

Malheureusement, nous ne connaissons pas l'origine des problèmes testés et les solutions obtenues. Par conséquent, il nous est impossible d'apprécier la difficulté de ces derniers et la qualité des solutions approchées trouvées.

Chapitre 3

Approches SIMD massif du parcours de graphes d'états

Nous avons vu dans les chapitres concernant les machines MIMD que deux des trois méthodes de parallélisation des algorithmes de parcours de graphes d'états, la décomposition des parcours de graphes d'états et le découpage des intervalles de recherche, sont des techniques intéressantes. Il nous reste à savoir si elles sont aussi applicables aux machines de type SIMD massif.

Nous nous avons également cherché à savoir si la troisième méthode de parallélisation, celle du traitement des états, peut ou non être rendue intéressante pour ce type de machine.

3.1 Architectures des machines SIMD massif utilisées

Dans la littérature, l'essentiel des algorithmes massivement parallèles ont été implanté sur la Connection Machine-2. Nous ferons donc un rapide rappel de l'architecture de cette machine.

Cependant les études théoriques effectuées ne perdent pas leur généralité pour d'autres machines de cette famille.

La Connection Machine-2 (CM-2) [Hillis 1985] de Thinking Machine Corporation (TMC) est issue d'un projet mené par Daniel Hillis au MIT dans les années 70. Elle possède les caractéristiques suivantes :

- c'est une machine de type SIMD et donc fortement *synchrone*;
- les processeurs élémentaires (p.e.) travaillent sur des données d'un bit;
- chaque p.e. dispose d'une *mémoire locale* de 256 kbits, soit 32 Koctets;
- 16 p.e. sont regroupés avec un circuit d'interconnexion (appelé *routeur*) sur une CM chip;
- 2 CM chips (32 p.e. + 2 routeurs) partagent un composant mémoire et un accélérateur en virgule flottante et ils forment une Sprint chip;
- les CM chips sont placés sur les nœuds d'un *réseau hypercubique* de degré 12 et assurent le routage des messages de communication entre les p.e.;

- il existe un bit de *context* par p.e. pour mettre les p.e. en position active;
- 4000, 8000 ou 16000 p.e. sont regroupés dans un *séquenceur*, et on peut mettre au maximum quatre séquenceurs par machine allant ainsi jusqu'à 64k de p.e. au maximum;
- la communication avec le monde extérieur et le lancement du flux d'instructions s'effectuent à travers un *frontal* qui est souvent une station de travail.

La configuration en hypercube des CM chips permet d'obtenir les propriétés intéressantes pour l'adressage des processeurs, la communication entre processeurs et l'extensibilité du réseau.

Propriété 4 *Un hypercube de degré N a les propriétés suivantes :*

- *il possède* 2^N *sommets,*
- *la taille de l'adressage des sommets est de N bits,*
- *les adresses de deux sommets voisins ont seulement un bit de différence,*
- *le code de Gray permet de trouver un cycle Hamiltonien dans le réseau de sommets,*
- *chaque sommet a exactement N voisins,*
- *le diamètre du réseau est égal à N,*
- *le réseau peut être étendu par puissance de 2 sommets.*

3.2 Problèmes posés par le parallèlisme SIMD massif

Le premier problème est celui introduit par l'architecture même des machines SIMD massivement parallèles. Le fonctionnement de ces machines sont de façon générale fortement *synchrones*. Elles sont particulièrement bien adaptées aux applications de type *partitionnement de données*[1] qui sont principalement numériques avec des structures de données régulières et souvent statiques (connues a priori). Or la méthode de décomposition des parcours de graphes d'états développe et explore de manière *asynchrone* des arbres ou des graphes *irréguliers* avec des structures de données irrégulières et dynamiques (élaborées en cours d'algorithme) comme nous avons pu les voir avec les machines MIMD.

Pour surmonter cette difficulté liée aux caractères incompatibles du fonctionnement des machines de type SIMD massif et des algorithmes de parcours de graphes d'états, il suffit de faire l'observation suivante. Les opérations de génération ou d'évaluation durant un parcours de graphe peuvent être effectuées indépendamment sur des états différents. Deux processus engendrant ou évaluant deux états différents ne communiquent pas entre eux et ils peuvent donc fonctionner en synchrone.

Cependant si les processus ne communiquent pas entre eux, nous risquerions d'avoir un surcoût de recherche important pour certains proce as et des situations de famine pour

[1] Data partitioning en anglais.

d'autres. Le fonctionnement synchrone des machines SIMD massif nous interdit d'avoir des phases d'exploration en même temps que des phases d'équilibrage des charges comme dans le cas des machines MIMD. Nous sommes donc amenés à effectuer des phases *alternées* et *distinctes* de recherche et d'équilibrage des charges.

Le deuxième problème est celui de la méthode de découpage des intervalles de recherche qui ne semble pas applicable puisque tous les processus doivent être dans le même intervalle de recherche. Mais nous pouvons envisager de regrouper des processus sur des intervalles de recherche différents.

Notamment sur la Connection Machine-2, il est possible de regrouper les processeurs élémentaires par séquenceur, et chaque séquenceur pourrait avoir un intervalle de recherche différent. Cependant, aucun des algorithmes existants n'utilise cette possibilité.

Le regroupement de processeurs élémentaires n'évite pas pour autant les situations de famine au début et vers la fin d'un intervalle de recherche. Une solution serait de permettre le commencement de la résolution d'un autre problème au même intervalle de recherche.

En troisième lieu, la parallélisation des traitements des états peut être bien adaptée aux machines SIMD massif. On peut imaginer l'utilisation de *pattern-matching* dans l'évaluation d'un état, par exemple l'évaluation d'une position aux échecs. Mais cette méthode est intéressante uniquement si le traitement d'un état peut être décomposé en parties indépendantes et identiques. Aussi, avec un grand nombre de processus disponibles, par exemple la Connection Machine-2, il faut que la taille de l'état à traiter soit grande, sinon, la famine peut être énorme.

Du fait de ces contraintes, les algorithmes proposés ne peuvent pas utiliser du tout cette méthode de parallélisation qui est pourtant à notre avis la mieux adaptée au fonctionnement de ce type de machine.

3.3 Les parallélisations à stratégie distribuée

L'absence de mémoire partagée de la machine cible CM-2 implique que dans les implantations des algorithmes présentés ci-dessous, toutes les parallélisations sont à stratégie distribuée et il n'existe pas de structures de données partagées.

3.3.1 Algorithme Parallel-Retracting A* [Evett 1990]

L'algorithme Parallel-Retracting A* (PRA*) est une parallélisation de l'algorithme Retracting A* (RA*) proposé par ces mêmes auteurs, version modifiée de A*.

Le principal défaut de l'algorithme A* [Nilsson 1980] est sa complexité spatiale exponentielle. Si l'algorithme A* doit développer une arborescence uniforme de degré b et de profondeur I pour trouver une solution optimale, alors la complexité spatiale correspondante est en $O(b^I)$. Cette complexité spatiale est ramenée à $O(b)$ avec l'algorithme IDA* [Korf 1985], mais ce dernier doit redévelopper à chaque itération tous les états depuis l'état initial.

L'algorithme RA* est un compromis entre A* et IDA*. L'idée essentielle est de contrôler la place mémoire utilisée de A* sans avoir à redevelopper les mêmes états plusieurs fois

de suite comme dans IDA*. Pour se faire, un paramètre K et une fonction d'évaluation auxiliaire e ont été définis.

Définition 3 *Le paramètre K est un seuil du nombre d'états de la file OUVERT de A*. Il est défini par l'inégalité*

$$K \geq d + b$$

où d est la longueur du chemin optimal, b le nombre d'états frères sur le chemin optimal, et $d + b$ le nombre d'états du chemin optimal (cf. Figure 3.1).

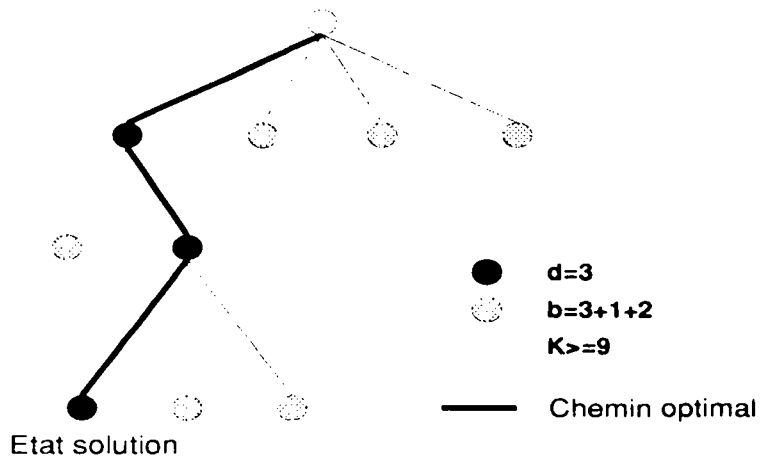

Figure 3.1 : Calcul du seuil K.

Définition 4 *La fonction d'évaluation auxiliaire e est définie comme suit.*
Soient j un état donné, i l'état du père de j, et k, l, m les états fils de j, alors la valeur de l'évaluation auxiliaire pour l'état j, $e(j)$ est égale au maximum entre l'évaluation de j et l'évaluation auxiliaire de i ($e(j) = \max\{f(j), e(i)\}$) si l'état j n'a pas encore été développé. Cette même évaluation, $e(j)$ est égale au minimum des évaluations auxiliaires des états k, l, m si l'état j a été développé ($e(j) = \min\{e(k), e(l), e(m)\}$, cf. Figure 3.2). La fonction d'évaluation f est toujours définie par la somme de deux fonctions, $g + h$.

En outre, on fait une hypothèse supplémentaire non limitative : la fonction d'heuristique h est supposée *monotone* (ou *admissible*). Cette hypothèse est non limitative dans la mesure où à partir d'une fonction heuristique non monotone, il est toujours possible d'en construire une monotone [Korf 1985].

Principe de l'algorithme PRA*

Le déroulement de l'algorithme RA* s'effectue de la façon suivante :

- développer l'arborescence de recherche suivant l'algorithme A* en gérant les files OUVERT et FERMER.

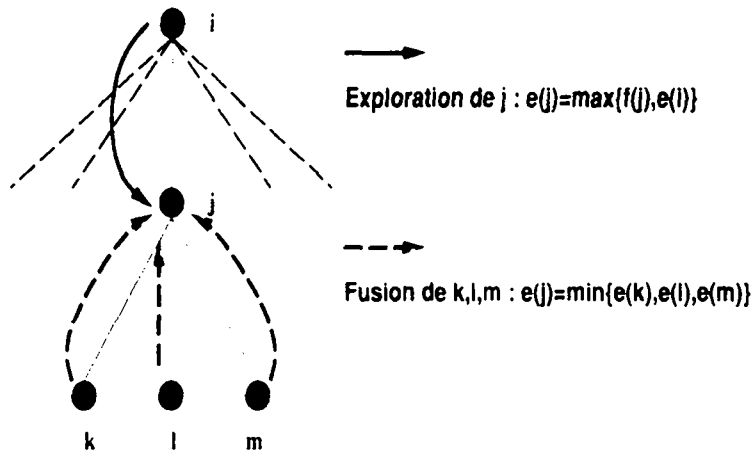

Figure 3.2 : La fonction d'évaluation auxiliaire e.

- si pendant le parcours, le nombre d'états dans OUVERT dépasse le paramètre K alors on *fusionne*[2] les états les moins prometteurs (de plus petites évaluations auxiliaires e), puis on va chercher les états pères de ces états non prometteurs dans la file FERMER pour les remettre dans la file OUVERT.

Il faut remarquer lorsque l'on remet des états pères dans la file OUVERT, les valeurs de la fonction auxiliaire des états pères sont égales aux minima des valeurs de la fonction auxiliaire des états fils non prometteurs.

Un exemple de déroulement est donné sur la Figure 3.3. Dans cet exemple, on a posé $K = 7$. La première fusion concerne les états N, O, P. Leur état père G est remis de la file FERMER (F) vers la file OUVERT (O) avec une valeur $e(G) = \min\{e(N), e(O), e(P)\} = 30$ alors que $e(G) = \max\{f(G), e(C)\} = 6$ lorsque les états fils n'étaient pas fusionnés. La deuxième fusion concernant les états non prometteurs K, L et leur état père E s'effectue exactement de la même façon.

```
O:A[3]<                                  F:<
O:B[4],C[5],D[7]<                        F:A[3]<
O:C[5],D[7],E[8],F[9]<                   F:A[3],B[4]<
O:G[6],D[7],H[7],E[8],F[9],I[10]<        F:A[3],B[4],C[5]<
O:D[7],H[7],E[8],F[9],I[10],N[30],O[35],P[50]<
F:A[3],B[4],C[5],G[6]<
O:D[7],H[7],E[8],F[9],I[10],G[30]<       F:A[3],B[4],C[5]<   Fusion
O:H[7],E[8],F[9],J[9],I[10],G[30]<
F:A[3],B[4],C[5],D[7]<
O:E[8],F[9],J[9],I[10],Q[20],G[30]<
F:A[3],B[4],C[5],D[7],H[7]<
O:F[9],J[9],I[10],Q[20],K[21],L[25],G[30]<
F:A[3],B[4],C[5],D[7],H[7],E[8]<
O:J[9],I[10],M[13],Q[20],K[21],L[25],G[30]<
```

[2] Retract en anglais.

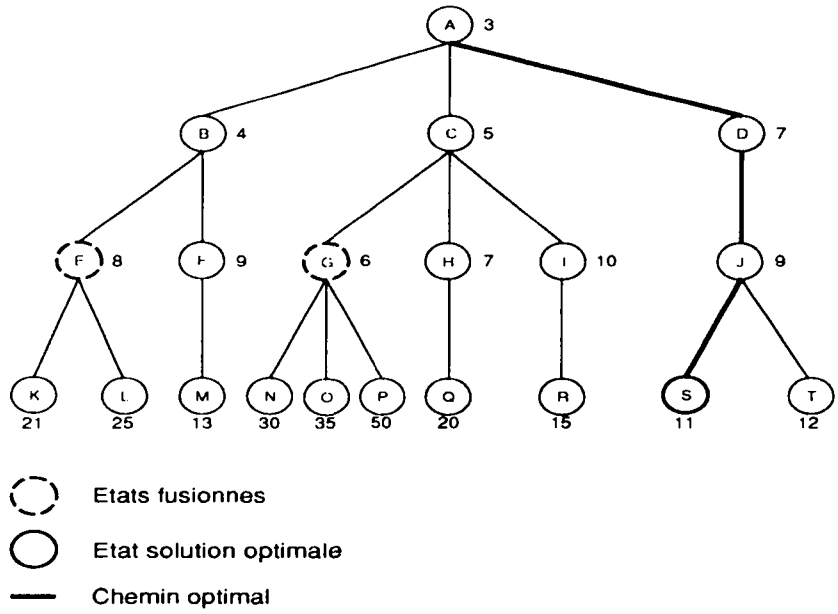

Figure 3.3 : Un exemple de déroulement de RA* avec $K = 7 \geq 3 + (2 + 0 + 1) = 6$.

```
F:A[3],B[4],C[5],D[7],H[7],E[8],F[9]<
O:I[10],S[11],T[12],M[13],Q[20],K[21],L[25],G[30]<
F:A[3],B[4],C[5],D[7],H[7],E[8],F[9],J[9]<
O:I[10],S[11],T[12],M[13],Q[20],E[21],G[30]<
F:A[3],B[4],C[5],D[7],H[7],F[9],J[9]<                    Fusion
O:S[11],T[12],M[13],R[15],Q[20],E[21],G[30]<
F:A[3],B[4],C[5],D[7],H[7],F[9],J[9],I[10]<
```

Dans l'algorithme PRA* implanté sur la CM-2, chaque processeur élémentaire exécute l'agorithme RA* et sélectionne à chaque itération le *meilleur état local* de sa file OUVERT locale. On est ainsi certain qu'une solution optimale sera sélectionnée.

Aprés qu'un état ait été sélectionné et développé, ses états fils engendrés sont communiqués aux autres processeurs élémentaires. Si durant ce développement, un p.e. n'a plus suffisamment de place mémoire disponible, il va fusionner les états les moins prometteurs de sa file locale en retrouvant leur état père. Ce dernier est ensuite communiqué à un autre p.e. pour libérer de la place mémoire.

La communication des états d'un p.e. vers les autres se fait par intermédiaire d'une *table hash globale*. Chaque p.e. est un *tiroir*[3] dans cette table, et l'évaluation des états donne avec la fonction hash le p.e. correspondant. Ainsi deux états identiques engendrés par deux p.e. différents sont communiqués au même p.e. qui n'explorera qu'une seule fois cet état. Une recherche est effectuée par chaque p.e. pour éliminer les doublons. Les surcoûts de recherche sont donc évités.

[3] Bucket en anglais.

Structures de données

Chaque p.e. gère dans sa mémoire locale un *nœud actif* et une *entrée* dans une table hash globale. Le nœud actif sert au développement d'un état à explorer.

L'entrée dans la table hash globale est une file de priorité qui permet aux p.e. de trier les états à explorer par ordre croissant de la fonction auxiliaire c. Cette file de priorité doit permettre au p.e. d'une part de sélectionner l'état le plus prometteur (de plus petite évaluation auxiliaire c), et d'autre part, d'effectuer les fusions des états les moins prometteurs.

Problèmes posés par le parallèlisme

D'abord le problème de la communication des états entre les p.e.. L'algorithme PRA* tel qu'il est présenté ci-dessus a une fréquence de communication très élevée, puisqu'il y a communications entre les p.e. lorsqu'il y a génération ou fusion d'états. Ce problème reste ouvert.

La fusion des états pose également un problème. Le mécanisme de la fusion des états impose la connaissance de leurs états pères et frères. Cette connaissance nécessite une gestion complexe des files de priorité locales. Malheureusement, les auteurs [Evett 1990] n'ont pas été très explicites sur ce point. On sait seulement que la sélection d'un état et l'élimination des doublons sont effectuées au prix d'un examen systématique du tiroir de chaque p.e..

Le troisième problème est celui de la définition de la fonction hash. Il semblerait qu'une *bonne* fonction hash soit assez difficile à trouver. Or une *bonne* fonction hash permet d'avoir une répartition équilibrée des états sur l'ensemble des p.e. disponibles. Les fusions d'états peuvent alors avoir lieu simultanément sur un grand nombre de p.e. et donc améliorer les performances de l'algorithme.

Résultats expérimentaux

Les tests ont été menés avec la taquin 4x4 dont les états initiaux sont dans [Korf 1985]. Les auteurs curieusement ne rapportent aucun résultat d'accélération, mais uniquement le nombre d'états explorés comparés à celui de IDA*. PRA* n'engendre que 38% des états engendrés par IDA*.

Cependant, PRA* nécessite énormément de communications entre les p.e.. Cet algorithme est par conséquent mieux adapté aux applications où les fonctions d'évaluation sont complexes. Les performances semblent également liées à la définition d'une bonne fonction hash.

Les auteurs espèrent améliorer les performances par 100 en apportant des corrections et des optimisations au programme. Les optimisations consistent essentiellement à utiliser les communications locales qui sont beaucoup moins coûteuses que les communications globales sur la CM-2.

3.3.2 Algorithme SIMD IDA* [Powley 1989, Powley 1991b, Powley 1991a]

L'équipe Powley et al. ont proposé une version SIMD massif de l'algorithme IDA* [Korf 1985] baptisée SIMD IDA*. Elle est semblable à celle de PWS proposée pour les machines de type MIMD sans mémoire partagée (section 2.4.2).

Le graphe d'états est parcouru avec l'algorithme A* pour obtenir des états-frontière. Une limite au nombre des états-frontière est fixée, soit arbitrairement, soit en fonction des ressources mémoires de la machine cible. Chaque état-frontière ainsi obtenu est affecté à un processeur et ce dernier explore son état avec l'algorithme IDA* jusqu'au seuil de recherche global.

Cependant, avec les machines de type SIMD, les processeurs ne peuvent fonctionner de façon asynchrone. La communication inter-processeurs ne peut donc être effectuée pendant le parcours du graphe d'états et vice-versa. Tous les processeurs sont toujours dans une phase d'exploration du graphe d'états ou dans une phase de communication. Or la communication est cruciale pour un bon équilibrage des charges évitant ainsi les situations de famine et de surcoût de recherche. Il importe donc de bien définir les phases de communication pour l'équilibrage des charges.

Powley et al. en ont proposé trois types : l'équilibrage à la *phase d'initialisation*, *entre les itérations* (des itérations de l'algorithme IDA*) et *pendant les itérations*. L'instant d'équilibrage des charges pendant les itérations peut être fixé de façon *statique* ou *dynamique*. Un modèle analytique pour un équilibrage optimal a d'ailleurs été proposé.

L'implantation de SIMD IDA* est faite aussi sur une CM-2. Les mesures de performances de SIMD IDA* sont effectuées suivant trois critères : l'*accélération* qui est mesurée par rapport au temps séquentiel estimé, l'*efficacité* qui est décomposée en quatre facteurs mesurables et l'*extensibilité* qui indique le comportement de l'algorithme face à l'augmentation du nombre de processeurs utilisés.

Toutes ces idées sont expliquées en détail dans les sections suivantes.

Principe de l'algorithme de SIMD IDA* sur la CM-2

L'algorithme SIMD IDA* peut être résumé en trois phases comme suit.

1ère phase : *A* parallèle*, génération des états et affectation de ces états aux p.e. disponibles jusqu'à saturation de la machine, on obtient alors des *états-frontière*.

$$Seuil_1 = \min_n f(n) \text{ où } n \text{ est un état.}$$

Les p.e. ayant des états-frontière n dont les valeurs $f(n)$ sont supérieures au $Seuil_1$ restent inactifs dans la 1ère phase de recherche (itération).
Remarque : $f(n) = Seuil_1$ ou $f(n) = Seuil_1 + 2$ pour le taquin.

TANT QUE une solution optimale n'est pas obtenue

2ème phase : Chaque p.e. effectue une recherche en *profondeur d'abord* (IDA*) jusqu'au $Seuil_i$ global.

Si tous les p.e. ont atteint le *Seuil*, global et qu'il n'y a pas de solution optimale, alors on passe à la 3ème phase.
Si une solution optimale est trouvée alors arrêt.

3ème phase : Equilibrage des charges (à la fin d'une itération), on *fusionne* les états-frontière de faible charge (estimée à l'itération précédente) et on développe les états-frontière de forte charge.

FIN TANT QUE.

La Figure 2.7 permet de visualiser les états-frontière et les algorithmes utilisés (A* et IDA*) pendant le parcours d'un graphe d'états.

La première phase est la phase d'initialisation de SIMD IDA*. L'algorithme A* est utilisé pour engendrer des états qui sont affectés aux p.e. disponibles au cours de la génération. Un p.e. possédant un état engendre les états fils. Le p.e. garde alors le premier état fils engendré et distribue les autres états aux p.e. disponibles. Cette phase a lieu jusqu'à saturation des p.e. disponibles. A l'instant de saturation, tous les p.e. ont un état. Ces états sont appelés des *états-frontière*. Si pendant cette première phase, une solution optimale a été trouvée, alors SIMD IDA* s'arrête. Sinon, les états-frontière sont alors explorés dans une deuxième phase avec l'algorithme IDA*. Le premier seuil de recherche $Seuil_1$ de IDA* est fixé au minimum des valeurs des états-frontière $f(n)$ où n est un état-frontière. Pour la première itération de IDA*, les états-frontière ayant une valeur $f(n)$ supérieure au $Seuil_1$ ne sont pas explorés, les p.e. correspondants restent inactifs en désactivant le bit de *contexte*.

La deuxième phase consiste à effectuer une recherche en *profondeur d'abord* avec l'algorithme IDA* jusqu'au $Seuil_i$ global. Pour la recherche en profondeur d'abord, chaque p.e. dispose d'une pile locale d'états. Chaque p.e. exécute l'algorithme IDA* sur son état-frontière jusqu'au seuil courant global. Si une solution optimale est trouvée alors on s'arrête, sinon on passe à la phase trois.

La troisième phase est une phase d'équilibrage des charges *entre les itérations*, nous verrons qu'il existe deux autres types d'équilibrage des charges. Pendant une itération de IDA*, la charge de travail de chaque p.e. sur son état-frontière est estimée par la taille du sous-graphe exploré jusqu'au seuil courant. Les états-frontière frères de faible charge (les tailles des sous-graphes explorés sont inférieures à la taille moyenne des sous-graphes) sont *fusionnés* et ils libèrent les p.e. correspondants. Tandis que ceux de forte charge (les tailles des sous-graphes explorés sont supérieures à la taille moyenne des sous-graphes) sont développés, et les nouveaux états-frontière ainsi engendrés sont affectés aux p.e. disponibles libérés par les états fusionnés. La phase deux et trois sont alors ré-itérées jusqu'à l'obtention d'une solution optimale.

Trois types d'équilibrage des charges

Outre l'équilibrage des charges *entre les itérations* par *fusion* et *développement* des états-frontière pendant la troisième phase de l'algorithme, il existe deux autres types d'équilibrage des charges.

La première est réalisée *à la phase d'initialisation*. Les états ayant les plus petites valeurs d'évaluation $f(n)$ sont d'abord développés. Ainsi à la fin de cette phase d'initialisation tous les états-frontières doivent avoir des valeurs $f(n)$ proches. Pour le taquin par exemple, nous aurons des valeurs comme $Seuil_1 = \min_n f(n)$, $Seuil_1 + 2$, $Seuil_1 + 4$ .... Cette façon de procéder a pour but d'obtenir des états-frontière ayant des sous-graphes à peu près de la même taille et donc de la même charge de travail.

La seconde est effectuée *pendant les itérations* de IDA*. Durant l'exécution de l'algorithme si le nombre de p.e. actifs est en-dessous d'un seuil, on ré-équilibre les charges en faisant transférer *temporairement* les états non explorés des piles locales des p.e. actifs vers les piles des p.e. inactifs. Les états à la base des piles locales (en haut des sous-graphes de recherche) sont transférés en premier, car ces états nécessitent en général des explorations plus longues et donc une charge de travail plus importante pour peu de transferts. Il faut noter que les affectations des états-frontière aux p.e. ne sont pas modifiées. Une fois qu'un p.e. a fini d'explorer les états d'un autre p.e., le premier va rendre les résultats au second.

Il reste maintenant à connaître le seuil des p.e. actifs en-dessous duquel il faut équilibrer les charges pendant les itérations.

Détermination du taux d'activité

Dans cette section, nous allons analyser la détermination du taux des p.e. actifs en-dessous duquel il est intéressant de ré-équilibrer les charges. Ce taux peut-être fixé de manière *statique* ou *dynamique*. Nous verrons que le deuxième choix est plus souple d'emploi et mieux adapté à la plupart des applications susceptibles d'utiliser l'algorithme SIMD IDA*. Avant d'aller plus loin, voyons d'abord quelques définitions préliminaires.

Définition 5 *On appelle* taux d'activité des p.e. X *avec* $X \in [0, 1]$, *le taux qui mesure à chaque instant le pourcentage du nombre de p.e. actifs.*

Définition 6 *Soit* P *le nombre de p.e. total et* $X * P$ *le nombre de p.e. actifs à chaque instant. On appelle* seuil d'activité *(mesuré en nombre de p.e. actifs), la valeur en dessous de* $X * P$ *un équilibrage des charges est déclenché*

Le seuil d'activité peut-être fixé de façon *statique*. Frye et Myczkowski [Frye 1991] ont trouvé que $2/3 * P$ est le meilleur seuil d'activité pour leurs applications. Par conséquent si le taux d'activité X est inférieur à 2/3 alors un équilibrage des charges a lieu.

Cependant le seuil d'activité statique pose un difficulté majeure. En effet, la détermination du seuil d'activité est fonction de la charge de travaux restants et du coût des équilibrages pendant les itérations. Si la charge (la durée d'exploration des états restants) est faible par rapport au coût des équilibrages (la durée nécessaire pour effectuer un équilibrage des charges), alors le seuil d'activité doit être petit et vice-versa. Or la charge de

travail diminue pour l'ensemble de p.e. à mesure que les recherches avancent, alors que le coût d'équilibrage reste constant. Donc, si la quantité de travail (le nombre d'états à explorer) est trop faible pour être répartie à l'ensemble des p.e. et que le seuil d'activité est grand, alors nous risquons d'avoir des équilibrages même s'ils ne sont pas nécessaires.

Powley et al. proposent de déterminer le seuil d'activité d'une manière *dynamique*.

Soit $A(t)$ le *nombre de p.e. actifs* à l'instant t, et $W(t)$ la *charge de travail des p.e.* pendant les t secondes d'une *phase de recherche* mesurée en secondes de p.e. actifs. Nous avons alors :

$$W(t) = \int_0^t A(t)dt.$$

Soit L le *coût du dernier équilibrage* en secondes. De ces mesures, nous déduisons :

$$\frac{W(t)}{t+L}$$

le taux moyen de la charge des p.e. d'une phase de recherche.
En dérivant, nous trouvons :

$$\frac{\partial}{\partial t}\frac{W(t)}{t+L} = \frac{W(t)}{-(t+L)^2} + \frac{W'(t)}{t+L}.$$

D'où

$$\frac{\partial}{\partial t}\frac{W(t)}{t+L} \leq 0 \Rightarrow \frac{W(t)}{t+L} \geq W'(t) = A(t),$$

et les charges doivent être équilibrées dès que

$$\frac{W(t)}{t+L} \geq A(t).$$

En clair, il faut toujours garder le taux moyen de la charge des p.e. *inférieure* aux p.e. actifs avec les équilibrages, pour obtenir un taux moyen maximal de charge des p.e. pendant une phase de recherche.

Les auteurs ont présenté également un modèle analytique pour la détermination du seuil d'activité dans leur article [Powley 1991a]. Ce modèle théorique permet de simuler les performances de l'équilibrage dynamique des charges entre les itérations.

Résultats expérimentaux : trois critères complémentaires

Les mesures de performances de l'algorithme SIMD IDA* ont été présentées suivant trois critères : *accélération*, *efficacité* et *modularité de croissance*.

La mesure de l'efficacité est cependant assez originale, elle n'est pas le rapport entre l'accélération sur le total de processeurs utilisés comme dans la définition habituelle. L'efficacité est décomposée en quatre paramètres mesurables que nous détaillerons plus tard.

Ces paramètres nous permettent de mieux apprécier le comportement de l'algorithme. Le dernier critère portant sur la modularité de croissance (scalability) de l'algorithme est essentiel pour connaître son comportement face à l'augmentation du nombre de processeurs utilisés.

L'implantation de SIMD IDA* est faite sur une CM-2 avec 16k de p.e.. Aussi, au lieu de résoudre les problèmes un par un, les auteurs proposent de résoudre les 100 [Korf 1985] en même temps. L'application test est le taquin 4x4.

Le critère d'accélération

Définition 7 *L'accélération est définie comme étant le rapport :*

$$T^n_{IDA*}/T^n_{SIMD\ IDA*}$$

où

- T^n_{IDA*} *est le temps de résolution de n problèmes pour la meilleure implantation de l'algorithme séquentiel IDA* sur la CM-2 avec un p.e..*
- $T^n_{SIMD\ IDA*}$ *est le temps de résolution des n mêmes problèmes par SIMD IDA* implanté sur la même machine avec 16k p.e..*

Dans la définition habituelle de l'accélération, $n = 1$.

Il est difficile de mesurer T^{100}_{IDA*}, le temps de résolution de 100 problèmes par la meilleure implémentation de l'algorithme séquentiel IDA* sur la CM-2 avec un p.e., car cela demanderait des années. Cependant, il est possible de l'estimer par le taux de génération d'états W^1_{IDA*}/T^1_{IDA*} où W^1_{IDA*} est le nombre d'états engendrés par IDA* pour un seul problème, et T^1_{IDA*} est son temps de résolution. On déduit alors

$$T^{100}_{IDA*} = \frac{W^{100}_{IDA*}}{W^1_{IDA*}/T^1_{IDA*}}.$$

Avec le calcul de T^{100}_{IDA*} et la mesure de $T^{100}_{SIMD\ IDA*}$, on obtient une accélération pour 100 problèmes égale à

$$S^{100} = \frac{T^{100}_{IDA*}}{T^{100}_{SIMD\ IDA*}} = \frac{9.7 \text{ années}}{10.8 \text{ heures}} = 7800 \text{ pour 16K p.e..}$$

Le critère d'efficacité

L'efficacité est mesurée via quatre paramètres définis ci-dessous.

Définition 8 *On appelle le taux d'accélération brut R le rapport :*

$$R = \frac{W^1_{SIMD\ IDA*,\ 1\ p.e.}/T^1_{SIMD\ IDA*,\ 1\ p.e.}}{W^1_{IDA*,\ 1\ p.e.}/T^1_{IDA*,\ 1\ p.e.}},$$

où 1 p.e. indique que ces mesures sont effectuées avec un seul p.e. de la CM-2 utilisée.

Ce taux R permet de mesurer la performance d'un algorithme parallèle en exécution séquentiel par rapport à un algorithme séquentiel.

Définition 9 *Le taux du temps de recherche par rapport au temps total F est défini par :*

$$F = \frac{T_W^{100}}{T_W^{100} + T_L^{100}}.$$

où T_W^{100} est le temps total passé dans les phases de recherche de l'algorithme SIMD IDA pour 100 problème, et T_L^{100} le temps total des équilibrages.*

F indique la proportion entre les phases de recherche (la résolution effective d'un problème), et les phases d'équilibrages. C'est un surcoût lié à l'utilisation du parallélisme.

Définition 10 *Le taux d'utilisation des p.e. pendant les phases de recherche U est :*

$$U = \frac{W_{SIMD\ IDA*}^{100}}{(\frac{T_{SIMD\ IDA*\ 1\ p.e.}^{100}}{T_{SIMD\ IDA*\ 1\ p.e.}^{100}}) * T_W^{100} * P}.$$

où P est le nombre de p.e. utilisés.

Ce taux U mesure le taux d'activité d'un p.e. d'une exécution parallèle à une exécution séquentielle.

Définition 11 *Le taux des nœuds engendrés par IDA* par rapport à SIMD IDA* est N :*

$$N = \frac{W_{IDA*}^1}{W_{SIMD\ IDA*}^1}.$$

Ce taux N permet de connaître le surcoût de recherche des états de SIMD IDA* par rapport à IDA*.

Avec ces quatre paramètres, l'efficacité E peut être définie comme

$$E = R * F * U * N.$$

A travers ces définitions, on constate que si le nombre d'équilibrages augmente, F diminue et U augmente. Or un équilibrage n'est efficace (E augmente) que si $F * U$ augmente, il ne sert donc à rien d'équilibrer trop de fois les charges si $F * U$ n'augmente pas.

Pour la résolution des 100 problèmes avec une CM-2 de 16k p.e., l'efficacité ainsi calculée est de

$$R * F * U * N = 0.772 * 0.879 * 0.839 * 1 = 57\%.$$

soit une accélération correspondante égale à 9300 pour 16k de p.e..

Il faut noter cependant que N vaut 0.836 en dans les mesures et non 1. Or les auteurs prétendent que le chiffre de 0.836 est biaisé par certaines instances de problèmes et que $N = 1$ est finalement plus proche de la vérité.

Le critère de modularité de croissance

Le critère de modularité de croissance est essentiel dans les mesures de performances d'un algorithme parallèle. Il nous permet de savoir si l'augmentation du nombre de processeurs utilisés ne ferait pas chuter les performances de l'algorithme de manière drastique.

Pour l'algorithme SIMD IDA*, l'augmentation du nombre de p.e. utilisés fait croître l'accélération si le problème à résoudre est difficile (nécessite l'exploration de nombreux états), car F et U augmentent. Si le problème est simple à résoudre, l'accélération stagne avec l'augmentation de p.e. utilisés.

Les auteurs pensent qu'ils peuvent optimiser les communications pour les équilibrages et d'obtenir des accélérations cinq fois supérieures.

Comparaison avec PRA*

Il est à noter que SIMD IDA* a un taux de génération d'états égal à 67 états par seconde par p.e., tandis que PRA* a un taux inférieur à un état par seconde par p.e.. Cependant, PRA* engendre nettement moins d'états que SIMD IDA*. Par conséquent, avec une application où la génération d'états est plus coûteuse que le cas du taquin 4x4, l'algorithme PRA* peut être plus intéressant.

3.3.3 Algorithme Iterative Deepening Parallel Search [Mahanti 1991]

L'idée de l'algorithme *Iterative Deepening Parallel Search* (IDPS) est aussi fondée sur l'algorithme IDA*. Nous allons simplement analyser ici les différences entre IDPS et SIMD IDA*, car ces deux algorithmes sont similaires.

Différences avec SIMD IDA*

Dans l'implantation de IDPS, un nœud est un état *entier*, alors que SIMD IDA* posséde uniquement l'état initial, et un nœud est simplement l'état initial avec toutes les modifications incrémentales permettant d'arriver à l'état courant.

L'algorithme IDPS n'effectue pas les équilibrages des charges *entre les itérations*. Mais un équilibrage a lieu dès qu'un p.e. devient inactif pendant les itérations. Par conséquent, il y a beaucoup d'équilibrages des charges.

Aussi pendant les équilibrages, IDPS peut communiquer plusieurs états indépendants à un seul p.e., tandis que SIMD IDA* ne transmet qu'un état à la fois ou plusieurs états issus d'un même état père.

Résultats expérimentaux

Cet algorithme est aussi implanté sur une CM-2 et a également pour application test le taquin 4x4.

Les résultats expérimentaux en comparaison avec ceux de SIMD IDA* sont les suivants. Sur le plan de l'efficacité, l'algorithme IDPS obtient 72% sur une machine de 16k p.e. pour les 100 problèmes [Korf 1985] contre 57% de l'algorithme SIMD IDA*. Avec une CM-2 de 8k p.e. sur les mêmes problèmes, l'algorithme IDPS atteint 92% contre 67% à l'algorithme

SIMD IDA*. Cependant il faut remarquer que ces pourcentages n'ont pas été calculés par rapport à une même implémentation séquentielle de IDA*.

En terme de durée de résolution avec une CM-2 de 8k p.e. pour 100 problèmes résolus, l'algorithme IDPS a nécessité 22.56 heures contre 21.66 heures à l'algorithme SIMD IDA*. Mais ce dernier génère 30% d'états de plus que l'algorithme IDPS.

Chapitre 4

Conclusion

Nous avons présenté en détail dans ce rapport divers algorithmes parallèles et distribués de parcours de graphes d'états en Intelligence Artificielle parallélisant soit A*, soit IDA*, soit une de leurs variantes. Ces algorithmes sont implantés sur une large gamme de machines, allant du MIMD à mémoire partagée au SIMD massif à mémoire distribuée, en passant par le MIMD à mémoire distribuée.

La plupart des parallélisations ont conduit à des résultats expérimentaux intéressants, mais limités. Des accélérations presque linéaires sont obtenues jusqu'à une vingtaine de processeurs. L'extension de ces résultats à un nombre plus important de processeurs reste à étudier, à la fois théoriquement et expérimentalement.

Une composante importante de l'efficacité des algorithmes de la famille de A* en séquentiel est la pertinence pour le problème de la fonction heuristique h. Elle a été peu étudiée dans le cadre de ces algorithmes parallèles.

La recherche actuelle, phénomène de mode ou non, tend à s'orienter vers l'étude des machines massivement parallèles. Quelques algorithmes ont été ainsi conçus pour la Connection Machine-2.

Il semble que la sortie des machines de type MIMD massivement parallèle (KSR, Paragon, ...) ait rendu moins coûteuses les machines de type SIMD massivement parallèle, et donc plus intéressantes pour les industriels. L'étude d'applications non numériques sur ce type de machine, surtout consacré aux applications numériques, devient alors justifiée.

Cependant, le parallélisme massif impose de résoudre des problèmes de définition et de gestion des structures de données parallèles (à accès concurrent ou réparties), d'équilibrage de charges et de communications entre les processseurs pour obtenir des accélérations linéaires.

D'autre part, l'essentiel des résultats des expériences présenté dans la littérature a été obtenu sur le jeu du taquin. Il reste donc à savoir quelles seraient les performances sur des applications plus proches des problèmes rencontrés dans la pratique comme les problèmes de chargement (sac à dos multidimensionnel), ou encore les problèmes de placement (allocation de processus à des processeurs).

Dans ces applications, une technique telle que celle du dépassement des seuils de recherche dans IDA*, est difficile à mettre en œuvre, car les valeurs de ces seuils sont a priori

inconnues.

L'étude de la parallélisation de ces algorithmes, qui nous conduit à réfléchir sur divers problèmes (structures de données à manipuler en parallèle, stratégie de régulation de charges, communication), dépasse ainsi le cadre strict d'algorithmes de parcours de graphes d'états en Intelligence Artificielle.

Elle peut s'étendre naturellement d'une part au parcours d'arborescence en Recherche Opérationnelle (Backtrack, Branch-and-Bound) et d'autre part, contribuer à l'analyse de problèmes classiques posés par la conception d'algorithmes parallèles.

Bibliographie

[Akl 1980] Akl (S.G.), Barnard (D.T.) et Doran (R.J.). Simulation and analysis in deriving time and storage requirements for a parallel alpha-beta algorithm. 1980, pp. 231-234.

[Baudet 1978] Baudet (G.M.). *The design and analysis of algorithms for asynchronous multiprocessors.* - Thèse de doctorat, Carnegie-Mellon University, 1978.

[Berliner 1989] Berliner (H. J.) et Ebeling (C.). - *Computers, Chess, and Cognition*, chap. 6 Hitech, pp. 79 109. - 1989.

[Campbell 1983] Campbell (Murray S.) et Marsland (T.A.). - A comparison of minimax tree search algorithms. *Artificial Intelligence*, vol. 20, n° 4, juillet 1983, pp. 347 367.

[Chakrabarti 1989] Chakrabarti (P. P.), Ghose (S.), Acharya (A.) et de Sarkar (S. C.). - Heuristic search in restricted memory. *Artificial Intelligence*, vol. 41, n° 2, 1989, pp. 197 221.

[Cung 1991] Cung (Van-Dat) et Roucairol (Catherine). - *Parcours Parallèle d'Arbres Minimax.* RR n° 1549, INRIA, novembre 1991. In French.

[Evett 1990] Evett (Matthew), Hendler (James), Mahanti (Ambujashka) et Nau (Dana). - Pra* : A memory-limited heuristic search procedure for the connection machine. *In : IEEE Frontiers of Massively Parallel Computation*, pp. 145 150. - octobre 1990.

[Ferguson 1988] Ferguson (Chris) et Korf (Richard E.). - Distributed tree search and its application to alpha-beta pruning. *In : Seventh National Conference Artificial Intelligence AAAI-88*, pp. 128 132. - août 1988.

[Finkel 1982] Finkel (R.A.) et Fishburn (J.P.). - Parallelism in alpha-beta search. *Artificial Intelligence*, vol. 19, 1982, pp. 89 106.

[Florin 1991] Florin (Gérard) et Lavallée (Ivan). - *La récursivité, mode de programmation distribuée.* RR n° 1536, INRIA, octobre 1991. In French.

[Frye 1991] Frye (Roger) et Myczkowski (Jacek). - Exhaustive search of unstructured trees on the connection machine. *Journal of Parallel and Distributed Computing*, 1991. Submitted to.

[Hillis 1985] Hillis (W. Daniel). *The Connection Machine.* Patrick Henry Winston and Michael Brady, the MIT Press, 1985.

[Hsu 1990] Hsu (Feng-Hsiung). *Large Scale Parallelization of Alpha-Beta search : An Algorithmic and Architectural Study with Computer Chess.* - Thèse de doctorat. Carnegie-Mellon University. School of Computer Science, février 1990.

[Ibaraki 1990] Ibaraki (Toshihide) et Katoh (Yoshiroh). Searching minimax game trees under memory space constraint. *Annals of Mathematics and Artificial Intelligence*, vol. 1, 1990, pp. 141-153.

[Kalé 1990] Kalé (L.V.) et Saletore (Vikram A.). - Parallel state-space search for a first solution with consistent linear speedups. *International Journal of Parallel Programming*, vol. 19, n° 4, 1990, pp. 251-293.

[Knuth 1975] Knuth (D.E.) et Moore (R.W.). An analysis of alpha-beta pruning. *Artificial Intelligence*, vol. 6, n° 4, 1975, pp. 293-326.

[Korf 1985] Korf (Richard E.). - Depth-first iterative-deepening : An optimal admissible tree search. *Artificial Intelligence*, no27, 1985, pp. 97-109.

[Kumar 1987] Kumar (Vipin), Ramesh (K.) et Rao (V. Nageshwara). - Parallel best-first search of state-space graphs : A summary of results. *The AAAI Conference*, 1987, pp. 122-127.

[Lecun 1991] Lecun (Bertrand), Mans (Bernard) et Roucairol (Catherine). - *Opérations concurrentes et files de priorité.* - RR n° 1548, INRIA, novembre 1991. In French.

[Li 1991] Li (Tao). - Parallel imprecise iterative deepening for combinatorial optimization. *International Journal of High Speed Computing*, vol. 3, n° 1, 1991, pp. 63-76.

[Mahanti 1991] Mahanti (Ambuj) et Daniels (Charles J.). - *SIMD Parallel Heuristic Search.* - Rapport technique n° UMIACS-TR-91-41, CS-TR-2633, College Park, Maryland, Computer Science Department, University of Maryland, mai 1991.

[Marsland 1982] Marsland (T.A.) et Campbell (M.). Parallel search of strongly ordered game trees. *ACM Computing Surveys*, vol. 14, n° 4, décembre 1982, pp. 533-551.

[Marsland 1985] Marsland (T.A.) et Popowich (F.). Parallel game-tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-7, n° 4, juillet 1985, pp. 442-452.

[Nau 1984] Nau (Dana S.), Kumar (Vipin) et Kanal (Laveen). - General branch and bound, and its relation to a* and ao*. *Artificial Intelligence*, vol. 23, 1984, pp. 29-58.

[Nilsson 1980] Nilsson (Nils J.). - *Principles of Artificial Intelligence.* Tioga Publishing Co., 1980.

[Pearl 1984] Pearl (Judea). - *Heuristics.* Addison-Wesley, 1984.

[Powley 1989] Powley (Curt) et Korf (Richard E.). Simd and mimd parallel search. *In : The AAAI symposium on Planning and Search.* mars 1989.

[Powley 1990] Powley (Curt), Ferguson (Christ) et Korf (Richard E.). - Parallel heuristic search : Two approaches. *In : Parallel Algorithms for Machine Intelligence and Vision*, pp. 42–65. - Kumar and Kanal, 1990.

[Powley 1991a] Powley (Curt), Ferguson (Chris) et Korf (Richard E.). Depth-first heuristic search on a simd machine. - juillet 1991. submitted to Artificial Intelligence.

[Powley 1991b] Powley (Curt), Ferguson (Chris) et Korf (Richard E.). - Parallel tree search on a simd machine. *In : The Third IEEE Symposium on Parallel and Distributed Processing.* - décembre 1991.

[Powley 1991c] Powley (Curt) et Korf (Richard E.). - Single-agent parallel window search. *IEEE Transactions on pattern analysis and machine intelligence*, vol. 13, n° 5, mai 1991, pp. 466–477.

[Rao 1987a] Rao (N. Nageshwara) et Kumar (Vipin). - Parallel depth first search, part i : implementation. *International Journal of Parallel Programming*, vol. 16, n° 6, décembre 1987, pp. 479–499.

[Rao 1987b] Rao (V. N.), Kumar (Vipin) et Ramesh (K.). - A parallel implementation of iterative-deepening-a*. *In : Proceeding AAAI*, pp. 178–182. - 1987.

[Rao 1987c] Rao (V. Nageshwara), Kumar (Vipin) et Ramesh (K.). - *Parallel Heuristic Search on Shared Memory Multiprocessors : Preliminary Results.* - Rapport technique n° AI85-45, Artificial Intelligence Laboratory, The University of Texas at Austin, juin 1987.

[Reinefeld 1985] Reinefeld (Alexander), Schaeffer (Jonathan) et Marsland (T. A.). - Information acquisition in minimal window search. *9th IJCAI*, vol. 2, 1985, pp. 1040–1043.

[Roucairol 1990] Roucairol (Catherine). - *Recherche arborescente en parallèle.* RR n° M.A.S.I. 90.4, Institut Blaise Pascal - Paris VI, 1990. In French.

[Schiper 1984] Schiper (André), Coray (Giovanni) et Hirsbrunner (Béat). A two-level control structure for parallel heuristic programming. *Recherche*, vol. 3, n° 1, 1984, pp. 28–37. - In french.

[Steinberg 1990] Steinberg (Igor) et Solomon (Marvin). Searching game trees in parallel. *In : International Conference on Parallel Processing*, pp. III–9 – III–17. 1990.

[Stockman 1979] Stockman (G.C.). A minimax algorithm better than alpha-beta ? *Artificial Intelligence*, vol. 12, 1979, pp. 179 196.

[Weill 1992] Weill (Jean-Christophe). The negac* search. – Université Paris 8-Saint Denis, personnal communication, 1992.

Unité de Recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

Unité de Recherche INRIA Lorraine Technopôle de Nancy-Brabois - Campus Scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 VILLERS LES NANCY Cedex (France)
Unité de Recherche INRIA Rennes IRISA, Campus Universitaire de Beaulieu 35042 RENNES Cedex (France)
Unité de Recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 GRENOBLE Cedex (France)
Unité de Recherche INRIA Sophia Antipolis 2004, route des Lucioles - B.P. 93 - 06902 SOPHIA ANTIPOLIS Cedex (France)

EDITEUR
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

ISSN 0249 - 6399